

UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

LÉO CARVALHO RAMOS ANTUNES

SEARCH-BASED HIERARCHICAL SOFTWARE MODULE CLUSTERING

Rio de Janeiro
August, 2026

UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

LÉO CARVALHO RAMOS ANTUNES

SEARCH-BASED HIERARCHICAL SOFTWARE MODULE CLUSTERING

Thesis presented to the Faculty of the Graduate Program in Informatics (PPGI) of the Federal University of the State of Rio de Janeiro (UNIRIO), in fulfillment of the requirements for the Ph.D. degree in Informatics.

Advisor: Márcio de Oliveira Barros

Rio de Janeiro
August, 2026

Catalogação informatizada pelo(a) autor(a)

A627 Antunes, Léo Carvalho Ramos

Modularização hierárquica de software baseada em busca / Léo Carvalho Ramos Antunes. – Rio de Janeiro : UNIRIO, 2026.
174 f.

Orientador: Márcio de Oliveira Barros.
Tese (Doutorado) – Universidade Federal do Estado do Rio de Janeiro, Programa de Pós-Graduação em Informática, 2026.

1. Engenharia de Software. 2. Modularização. 3. Clusterização Hierárquica. I. Barros, Márcio de Oliveira, orient. II. Título

Léo Carvalho Ramos Antunes

SEARCH-BASED HIERARCHICAL SOFTWARE MODULE CLUSTERING

THESIS PRESENTED TO THE FACULTY OF THE GRADUATE PROGRAM IN INFORMATICS (PPGI) OF THE FEDERAL UNIVERSITY OF THE STATE OF RIO DE JANEIRO (UNIRIO), IN FULFILLMENT OF THE REQUIREMENTS FOR THE PH.D. DEGREE IN INFORMATICS. APPROVED BY THE EXAMINING COMMITTEE SIGNED BELOW.

Approved by

Documento assinado digitalmente
gov.br MARCIO DE OLIVEIRA BARROS
Data: 02/10/2026 07:45:49-0300
Verifique em https://validar.iti.gov.br

Prof. D.Sc. Márcio de Oliveira Barros (Advisor)

Documento assinado digitalmente
gov.br CARLOS EDUARDO RIBEIRO DE MELLO
Data: 08/10/2026 23:23:50-0300
Verifique em https://validar.iti.gov.br

Prof. D.Sc. Carlos Eduardo Ribeiro de Mello

Documento assinado digitalmente
gov.br GLEISON DOS SANTOS SOUZA
Data: 13/10/2026 18:10:33-0300
Verifique em https://validar.iti.gov.br

Prof. D.Sc. Gleison dos Santos Souza

Documento assinado digitalmente
gov.br JERFFESON TEIXEIRA DE SOUZA
Data: 14/10/2026 07:28:27-0300
Verifique em https://validar.iti.gov.br

Prof. D.Sc. Jerffeson Teixeira de Souza

Documento assinado digitalmente
gov.br LEONARDO GRESTA PAULINO MURTA
Data: 16/10/2026 10:39:08-0300
Verifique em https://validar.iti.gov.br

Prof. D.Sc. Leonardo Gresta Paulino Murta

Rio de Janeiro
August, 2026

ABSTRACT

Antunes, Léo Carvalho Ramos. **Search-Based Hierarchical Software Module Clustering**. August, 2026. 160 pages. Doctoral Thesis. Graduate Program in Informatics, Federal University of the State of Rio de Janeiro (UNIRIO), Rio de Janeiro.

Heuristic search algorithms have been applied in various areas of Software Engineering, such as requirements prioritization, software architecture improvement, and time and cost planning. The research field of Search-based Software Engineering (SBSE) proposes the use of heuristic search techniques to solve Software Engineering challenges, which are described as optimization problems.

One of the challenges frequently analyzed from the SBSE perspective is the Software Module Clustering (SMC) problem. This problem concerns the distribution of modules (or classes) within a software project across larger structures called clusters (or packages). A good distribution of classes into packages tends to make software easier to develop and maintain. In addition, practice and experimental studies have shown a strong correlation between poor distributions and the presence of component failures.

The quality of the distribution of software modules into clusters is evaluated through metrics that quantify desired characteristics. The main factors used to assess the quality of distributions are coupling and cohesion. However, research shows that optimization based on metrics capturing structural characteristics alone does not yield an adequate distribution of classes across packages from the perspective of software developers. One reason that might have led to such limited results is the search for non-hierarchical class distributions across packages. Another reason may be the implicit relationships among classes that lack explicit structural connections.

In this thesis, we analyze an alternative approach to software modularization, aiming to achieve two distinct goals. First, we examine whether using a hierarchy-based metric (MDL) to guide the clustering algorithm yields solutions closer to those proposed by software developers. We conducted a study that shows improvements relative to a purely flat metric (MQ). Second, we extend the metric by considering inter-class distances based on how often classes change together in version-control commits, capturing implicit dependencies among classes in a software project. Likewise, we evaluate the results against the solutions proposed by developers and demonstrate further improvements over prior metrics. These results indicate that co-change relationships extracted from commits are a relevant source of information for the SMC problem, complementing the structural dependencies traditionally used to address it.

Keywords: Software Engineering, Software Modularization, Hierarchical Clustering.

RESUMO

Antunes, Léo Carvalho Ramos. **Modularização Hierárquica de Software baseada em Busca**. August, 2026. 160 f. Tese de Doutorado. Programa de Pós-Graduação em Informática (PPGI), Universidade Federal do Estado do Rio de Janeiro (UNIRIO), Rio de Janeiro.

Algoritmos de busca heurística têm sido aplicados em diversas áreas da Engenharia de Software, como a priorização de requisitos, a melhoria da arquitetura de projetos de software e o planejamento de tempo e custo de desenvolvimento dos projetos. O campo de pesquisa da Engenharia de Software Baseada em Buscas (Search-based Software Engineering, ou SBSE) propõe a utilização de técnicas de busca heurística para resolver problemas de Engenharia de Software descritos sob uma perspectiva de otimização.

Um dos problemas frequentemente analisados segundo esta perspectiva é o da Clusterização de Módulos de Software (CMS). Este problema lida com a distribuição de módulos (ou classes) de um software entre estruturas maiores chamadas *clusters* (ou pacotes). Uma boa distribuição das classes em pacotes tende a tornar o software mais fácil de desenvolver e manter. Além disso, estudos práticos e experimentais mostraram uma forte correlação entre distribuições ruins e a presença de falhas nos componentes.

A qualidade da distribuição de módulos de software em *clusters* é avaliada através de métricas que quantificam as características desejadas (*e.g.* coesão e acoplamento). No entanto, pesquisas mostram que otimizações baseadas em métricas que captam somente características estruturais não resultam em uma distribuição adequada do ponto de vista dos desenvolvedores de software. Duas razões que podem ter levado a resultados insatisfatórios são: distribuições não hierárquicas de classes em pacotes e a desconsideração de relacionamentos implícitos entre as classes do sistema.

Nesta tese, analisamos duas abordagens alternativas de clusterização. Primeiro, avaliamos se o uso de uma métrica hierárquica (*MDL*) produz soluções mais próximas às propostas pelos desenvolvedores. Realizamos um estudo que indica melhora em relação à métrica puramente estrutural (*MQ*). Em segundo lugar, estendemos a métrica considerando diferentes distâncias entre classes, com base na frequência com que elas aparecem juntas em commits de sistemas de controle de versão – permitindo capturar dependências implícitas entre elas. De igual modo, os resultados indicam melhora em relação às métricas anteriores e que as relações de co-modificação extraídas dos commits são uma fonte de informação relevante para o problema de CMS, complementar às dependências estruturais tradicionalmente utilizadas.

Palavras-chave: Engenharia de Software, Modularização, Clusterização Hierárquica.

LIST OF FIGURES

1.1 A HMD and its module tree (Lutz, 2001). 6

2.1 A HMD and its module tree (Lutz, 2001). 22

3.1 Number of packages (*DEV* x *MQ* x *MDL*). 49
3.2 Number of packages (*DEV* x *MQ* x *MDL*) boxplot. 51
3.3 Average number of classes per package (*DEV* x *MQ* x *MDL*). 52
3.4 Average number of classes per package (*DEV* x *MQ* x *MDL*) boxplot. 53
3.5 Maximum depth (*DEV* x *MQ* x *MDL*) 54
3.6 Maximum depth (*DEV* x *MQ* x *MDL*) boxplot. 55
3.7 $MojoFM$ (*MQ* x *MDL*) 56
3.8 $MojoFM$ value (*MQ* x *MDL*) boxplot. 57
3.9 Single-package commits ratio (*DEV* x *MQ* x *MDL*) 59
3.10 Single-package commits ratio (*DEV* x *MQ* x *MDL*) boxplot. 59
3.11 Average of packages per commit (*DEV* x *MQ* x *MDL*) 61
3.12 Average of packages per commit (*DEV* x *MQ* x *MDL*) boxplot. . . . 61
3.13 JGH *DEV* distribution. 62
3.14 Distribution proposed by the *MQ* optimization for JGH. 63
3.15 Distribution proposed by the *MDL* optimization for JGH. 63
3.16 *JMA DEV* distribution. 65
3.17 Distribution proposed by the *MDL* optimization for *JMA*. 65
3.18 Part of the *DEV* distribution for *JMA* (parameter-related classes). . . 66
3.19 Part of the *MDL* distribution for *JMA* (parameter-related classes). . . 67

4.1 The hypothetical software system used in the following sections to illustrate the proposed approach. 72
4.2 Example of a Commit Relation Graph (CRG) with eight classes. . . . 74
4.3 Example of an enhanced *HMD*. 80
4.4 Comparison between the authoritative decomposition (*DEV*) and an algorithmic solution (*HMD*). 87
4.5 Comparison between the authoritative decomposition tree (*DEV*) and an algorithmic solution tree (*HMD*). 88
4.6 Boxplots illustrating the ratio of the number of packages generated by *MQ*, *HMD*, and HMD_p^+ relative to the *DEV* baseline. 97
4.7 Heatmap illustrating the proportional ratio and deviation of generated packages relative to the *DEV* baseline. 98
4.8 Tree Edit Distance (*DEV* x *MQ* x *HMD* x HMD_p^+) boxplot. 101

4.9 Tree Edit Distance ratio (Movements / Number of Classes) distribution across configurations. 103
4.10 Tree Edit Distance ratio (Movements / Number of Classes) heatmap, illustrating relative algorithmic performance per instance. 104
4.11 Single-package commits ratio boxplot, illustrating the proportional ratio generated by MQ, HMD, and HMD_p^+ relative to the DEV baseline. 109
4.12 Single-package commits ratio heatmap, illustrating the proportional ratio of algorithmic localization relative to the DEV baseline. 110
4.13 Average number of packages per commit boxplot, illustrating the proportional dispersion ratio generated by MQ, HMD, and HMD_p^+ relative to the DEV baseline. 114
4.14 Average number of packages per commit heatmap, indicating algorithmic dispersion relative to the DEV baseline (Ratio < 1.0 indicates improvement). 115

LIST OF TABLES

3.1 Investigated software systems 45
3.2 Investigated systems/sub-modules ordered by the number of classes. . 46
3.3 The distribution of the runtime for the optimization based on MQ, considering the percentage of time running HAC and ILS_CMS. . . . 47
3.4 Number of packages (DEV x MQ x MDL). 50
3.5 Average number of classes per package (DEV x MQ x MDL). 52
3.6 Maximum depth (DEV x MQ x MDL). 55
3.7 $MojoFM$ value (MQ x MDL). 56
3.8 Percentage of one-package commits (DEV x MQ x MDL). 58
3.9 Average of clusters per commit (DEV x MQ x MDL). 60

4.1 Adjacency matrix of an example graph with eight classes. 75
4.2 Adjacency matrix one-dimensional array for example graph with eight classes (upper triangular form). 77
4.3 Example of Jaccard similarity calculation between two classes. 79
4.4 Jaccard similarity values between classes, grouped by CRG. 82
4.5 ZSS Transformation Operations ($HMD \rightarrow DEV$) 88
4.6 Class-to-Package mapping for the DEV and HMD decompositions. . 89
4.7 Commit-by-Commit analysis for the DEV and HMD decompositions. 90
4.8 Investigated software systems 92
4.9 Investigated systems/sub-modules ordered by the number of classes. . 93
4.10 Number of packages for the distributions of classes into packages produced by different algorithms for the selected instances. 96
4.11 Tree Edit Distance (ZSS) Comparison 100
4.12 Tree Edit Distance (ZSS) Comparison 102
4.13 Pairwise comparisons of the normalized Tree Edit Distance (ZSS/N) across configurations (Wilcoxon signed-rank test, Holm-corrected, $n = 21$) 103
4.14 Single-package commits ratio 108
4.15 Average number of packages per commit 112
4.16 Correlation (Pearson, n = 21) between the normalized structural improvement ($\Delta ZSS/N$) and system characteristics. 117
4.17 Performance of the distribution of classes to packages proposed by JodaMoney's authors, the distributions proposed by the optimization algorithms, and the distributions proposed by the subjects as of the number of single-package commits. 128

4.18 Tree distance between the distribution of classes to packages by the optimization algorithms and subjects to the distribution proposed by JodaMoney's authors. 129

LIST OF ABBREVIATIONS AND ACRONYMS

HAC	Hierarchical Agglomerative Clustering
HAC*	Hierarchical Agglomerative Clustering Star
ACDC	Algorithm for Comprehension-Driven Clustering
AEP	Google App Engine Plugin Core
APC	Average number of packages affected per commit
ATMR	ATtributes to Methods Ratio
CF	Cluster Factor
CRG	Commit Relation Graph
DEV	Developer-authored distributions
EC	External Coupling
ESC	Elastic Search Core
ESG	Elastic Search Geo
ESN	Elastic Search Native
EVM	EValuation Metric
HAC	Hierarchical Agglomerative Clustering
HMD	Hierarchical Modular Decomposition
ILS	Iterated Local Search
JCC	JavaCC
JGE	JavaGeom
JGH	JGit HTTP
JGL	JGit LFS
JGP	JGit PGM
JGS	JGit SSH
JJE	JUnit Jupiter Engine
JMA	JMetal Auto

JMC	JMetal Component
JML	JMetal Lab
JMO	Joda Money
JPE	JUnit Platform Engine
JPL	JUnit Platform Launcher
LJA	Log4J API
LJI	Log4J IO Streams
LJJ	Log4J JPA
LJL	Log4J Layout
LNS	Large Neighborhood Search
LUC	Lucene Codecs
LUH	Lucene Highlighter
LUN	Lucene Nori
MDG	Module Dependence Graph
MDL	Minimum Description Length
MeCl	Merge Clusters
Mojo	Moves and Joins
MojoFM	Moves and Joins eFfectiveness Measure
MQ	Modularization Quality
NAC	Numbers Among Classes
NP	Number of packages
SBSE	Search-based Software Engineering
SPCR	Single-package commits ratio
ZSS	Zhang and Shasha's Tree Edit Distance

CONTENTS

1 INTRODUCTION 1
1.1 Context and motivation 1
1.2 Problem characterization 2
1.3 Objectives and research proposal 7
1.4 Research questions 8
1.5 Contributions 9
1.6 Research methodology 11
1.7 Thesis organization 13

2 BACKGROUND AND RELATED WORK 15
2.1 Information Theory 15
2.1.1 The MDL principle and software module clustering 17
2.1.2 Length of Code Words with Known Probabilities 18
2.1.3 Ψ: The Complexity of a Design Graph 20
2.2 Formal Representation of Hierarchical Decompositions 22
2.3 Non-hierarchical Software Module Clustering Techniques 24
2.4 Hierarchical Software Module Clustering Techniques 31
2.5 Change History and Logical Coupling 34
2.6 Semantic and LLM-based Approaches 36
2.7 Final remarks 37

3 COMPARING HIERARCHICAL AND FLAT CLUSTERING OF SOFTWARE MODULES 39
3.1 Introduction 39
3.2 An algorithm for hierarchical software clustering 40
3.3 Fine-tuning the proposed algorithm 44
3.4 Analysis of different optimization drivers 48
3.4.1 Number of packages 49
3.4.2 Average number of classes per package 50
3.4.3 Maximum depth 53
3.4.4 *MojoFM* 54
3.4.5 Single-package commits ratio 57
3.4.6 Average of packages per commit 58
3.4.7 Visual comparison 62
3.5 Threats to validity 66
3.6 Final remarks 69

4 HIERARCHICAL CLUSTERING USING COMMIT DATA 71
4.1 Introduction 71
4.2 The Commit Relation Graph (CRG) Model 73
4.2.1 Identifying relationships from the project's change history 73
4.2.2 Representing the Commit Relation Graph 75
4.2.3 Calculating the weights of the edges 76
4.3 Integrating CRG into the MDL-based clustering algorithm 79
4.4 Optimization over the HMD_p^+ configuration 83
4.5 Quantitative evaluation of the approach 83
4.5.1 Research Questions and Metrics 84
4.5.2 Illustrative Example: Metrics Calculation 86
4.5.3 Problem Instance Selection 91
4.5.4 Comparative Analysis 94
4.5.5 Discussion 116
4.5.6 Threats to Validity 118
4.6 Developer-based Study 120
4.6.1 Subject characterization 121
4.6.2 Solution characteristics 123
4.6.3 Predictions on issue resolution 125
4.6.4 Predictions on concomitant changes 126
4.6.5 Comparison among solutions 128
4.6.6 Discussion 129
4.6.7 Threats to Validity 130
4.7 Summary 131

5 CONCLUSIONS 133
5.1 Overview 133
5.2 Addressing the Research Questions 134
5.3 Limitations 135
5.4 Future Work 136

REFERENCES 142

APPENDIX A LIST OF EXAMPLE COMMITS OF A HYPOTHETICAL SYSTEM 156

1 INTRODUCTION

This chapter presents our research motivation and the problem to be addressed. Furthermore, it highlights the research questions, expected contributions, and methodology. Finally, it presents the thesis proposal organization.

1.1 Context and motivation

Heuristic search algorithms have been applied in various areas of Software Engineering, such as requirements prioritization (Bagnall et al., 2001; Durillo et al., 2011; Xuan et al., 2012), software architecture improvement (Barros, 2014; Bowman et al., 2010; Praditwong et al., 2011), and time and cost planning (Aguilar-Ruiz et al., 2001; Antoniol et al., 2005; Burgess et al., 2001; Minku et al., 2013). Traditional search techniques, such as brute force or branch-and-bound (Lawler et al., 1966), may be used in some cases. However, as the addressed problems grow in size and complexity, a search procedure based on the exhaustive examination of all possible solutions may require too much processing power to find the optimal solution. Thus, using thorough searches in some situations, such as those requiring quick decision-making responses or the use of frequently updated data, becomes unfeasible.

The research field of Search-based Software Engineering (SBSE) (Arcuri et al., 2011; Bate et al., 2011; Fraser et al., 2018; Harman; Jones, 2001) proposes the use of heuristic search techniques to solve Software Engineering challenges, which are described as optimization problems. These techniques are more efficient in terms of processing power than traditional search, although they cannot be guaranteed to find optimal solutions to the analyzed problem. In scenarios where a complete search is not feasible, the goals of optimization techniques are relaxed, and the search seeks

reasonable (instead of optimal) solutions to the problem. Research has shown that this approach is promising for various Software Engineering problems.

One of the challenges frequently analyzed from the SBSE perspective is the Software Module Clustering (SMC) problem (Mitchell; Mancoridis, 2006). This problem deals with the distribution of the modules comprising a software project among larger structures called clusters. The modules considered in the organization of the software project might include classes, methods, or variables. The relationships between these modules represent the import of classes, the invocation of methods, or access to variables. Clusters are groupings of modules and represent packages, namespaces, or software subsystems. They are structures with a higher level of abstraction than software modules. This manuscript assumes that object-oriented programming is used, modules depict **classes**, and clusters denote **packages**. Dependencies between modules represent either importing classes, calling methods, or accessing variables. The pairs of terms $module|class$ and $cluster|package$ will be used for the same purpose in this manuscript.

A good distribution of classes into packages helps identify the classes responsible for implementing a feature, supports navigation between software components, and makes it easier to understand the software's structure (Gibbs et al., 1990; Larman, 2001; McConnell, 2004; Bavota; Gethers, et al., 2014). Therefore, a proper distribution of classes into packages tends to make software development and maintenance easier. In addition, practice and experimental studies have shown a strong correlation between poor distributions of classes into packages and the presence of component failures (Briand et al., 1999).

1.2 Problem characterization

The non-hierarchical SMC problem can be formally defined as follows. Given a set N with n modules, $N = \{M_1, M_2, M_3, ..., M_n\}$, the clustering problem consists of obtaining a set C containing k clusters, $C = \{C_1, C_2, ..., C_k\}$, each holding a subset of N, $C_i = \{m_j | m_j \in N\}$, such that C meets the following conditions:

$$\cup_{i=1}^{k} C_i = N \tag{1.1}$$

$$C_i \cap C_j = \emptyset \quad where\ 1 \leq i,\ j \leq k \text{ and } i \neq j \tag{1.2}$$

$$C_i \neq \emptyset \quad where\ 1 \leq i \leq k \tag{1.3}$$

Definition 1.1 indicates that the union of all k clusters gives us the set N, i.e., the n modules contained in N are distributed among k clusters in C. In addition, Definition 1.2 states there is no intersection between clusters, i.e., each module m belongs to one and only one cluster. Finally, Definition 1.3 says there are no empty clusters, i.e., a cluster needs to have at least one module. The non-hierarchical structure of a software system S, with its modules and their relationships, is usually represented as a directed graph called a Module Dependence Graph (MDG) (Mancoridis; Mitchell; Chen, et al., 1999). Formally, $MDG = \langle N, R \rangle$, where N is the set of modules and $R \subseteq N \times N$ is the set of ordered pairs $\langle u, v \rangle$ that represent the dependencies between modules u and v.

Regarding computational complexity, the SMC problem is of the NP-Hard class (Garey et al., 1979) and is a graph partitioning problem. Furthermore, when the number of clusters into which the software modules may be distributed is unknown beforehand, it represents a significant increase in the number of possible solutions compared to clustering problems in which the number of clusters is given.

The quality of the distribution of software modules into clusters is evaluated through metrics that quantify the desired characteristics. The most frequently used metrics to assess the quality of distributions are coupling and cohesion (Yourdon et al., 1979). Coupling measures the dependency between software modules (or the modules within a cluster) within a system. At the same time, cohesion measures how much the modules in a given cluster are dedicated to a single objective. Usually, coupling is measured by the number of dependencies between modules belonging to different clusters, while cohesion is calculated through the dependencies between modules

belonging to the same cluster. In general, the goal of clustering is to distribute the software modules into a given number of clusters to minimize coupling and maximize cohesion, with the restriction that each module must belong to a single cluster.

Mitchell and Mancoridis (2006) defined the MQ (Modularization Quality) metric, which attempts to balance the cohesion and coupling of clusters in a project to achieve a cohesive structure that is simultaneously loosely coupled. Harman et al. (2005) proposed the use of an alternative measure called EVM (EValuation Metric) (Tucker et al., 2001) for the SMC problem. This metric focuses on the cohesion of clusters, assuming that a project with cohesive clusters will have a relatively small number of dependencies between those clusters.

In addition to coupling and cohesion, it is possible to cite other metrics used to assess the quality of the distribution of modules in clusters, such as NAC (Numbers Among Classes), EC (External Coupling), and ATMR (ATtributes to Methods Ratio) (Simons et al., 2012), which assess the symmetry and elegance of the software, rewarding a more uniform distribution of modules. Similarity metrics such as MoJo (Tzerpos et al., 1999) and EdgeSim (Mitchell; Mancoridis, 2008) can also be used in clustering. These metrics make it possible to compare different solutions proposed to the same clustering problem, evaluating how close solution A is to solution B (considered ideal) and thus making it possible to compare the effectiveness of different clustering algorithms.

However, research shows that optimization based on metrics that capture only the structural characteristics of a software project does not result in an adequate distribution of modules into clusters from the point of view of software developers (Barros et al., 2015). Such optimization frequently leads to a large number of clusters, each with a single or a couple of strongly coupled modules. These clusters are usually connected through a dense network of cross-cluster dependencies, which adds to the complexity of the distribution.

This behavior reflects how these metrics balance cohesion and coupling. Minimizing coupling alone would lead to the trivial solution of a single cluster containing all modules; therefore, metrics such as MQ combine both aspects in a single

value. MQ sums the scores of all clusters, each depending on the ratio of dependencies internal to the cluster to those crossing its boundaries (Section 2.3). Thus, MQ does not directly minimize the total number of dependencies between clusters and may reward solutions with many small clusters, each grouping a few strongly connected modules – a behavior also observed in the experiments reported in Chapters 3 and 4. Multi-objective formulations that treat cohesion, coupling, and other cluster properties as separate objectives have also been proposed (Praditwong et al., 2011; Barros, 2012). Still, optimization models based on these structural measures tend to produce designs that developers would hardly accept (Barros et al., 2015). This suggests that the limitation lies not only in how cohesion and coupling are combined, but in relying exclusively on them, which motivates the alternative metric and the additional source of information investigated in this thesis.

One of the reasons that might lead to such limited results is the search for non-hierarchical distributions; that is, clusters can contain only modules. In such distributions, clusters cannot contain other clusters. On the other hand, developers commonly organize packages hierarchically, grouping related packages under broader ones. In Java – the language of the systems analyzed in this thesis –, such nesting has no semantics in itself: it is essentially a naming convention, and a sub-package has no special access to the classes of its parent package, nor vice versa (Gosling et al., 2023). Still, the hierarchy is reflected in package names, in the directory structure of the source code, and in the way development environments present the system. Therefore, the benefits expected from a hierarchical organization are cognitive rather than enforced by the language: it may help developers navigate and understand the system by grouping related packages into progressively broader concerns. Although this thesis does not directly evaluate these benefits, they motivate our investigation of hierarchical distributions of modules into clusters, which are compared with the hierarchical designs adopted by developers in Chapters 3 and 4.

The representation of the SMC problem needs to be modified to cope with hierarchical structures. For this purpose, Lutz (2001) proposed the Hierarchical Modular

Decomposition (HMD) representation. Such a representation describes a given system S as $HMD_S = \langle T_S, E_S \rangle$, where T_S is a module tree and E_S is a collection of dependencies between the modules. The leaf nodes in T_S are the modules of S, and the "internal nodes" in T_S correspond to the clusters. Instead of conveying only modules, a cluster may contain other clusters. E_S is identical to the dependencies set R in the MDG representation. Figure 2.1 shows an HMD and its module tree.

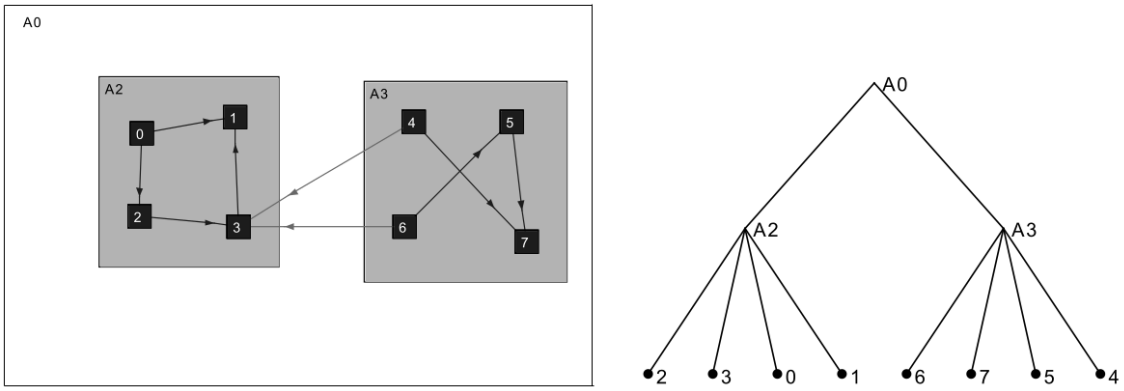


Figure 1.1: A HMD and its module tree (Lutz, 2001).

Wood (1998) developed a complexity measure for the hierarchical clustering of software projects based on a message-passing metaphor called the Minimum Description Length (MDL) principle. The metric makes no explicit mention of the concepts of cohesion and coupling. Still, it aims to permit trade-offs between engineering attributes (such as cohesion, coupling, and module size) to achieve a better structure in a hierarchical framework. It assumes the best distribution of modules into clusters for a software project is the "simplest" according to the MDL principle[1].

Both MDGs and HMDs can contain weighted and unweighted edges. The studies carried out in the context of this thesis are limited to unweighted edges, i.e., all the dependencies have the same weight.

[1] The MDL principle will be presented and discussed in Chapter 2.

1.3 Objectives and research proposal

In this thesis, we analyze an alternative approach to software modularization, aiming to achieve two different goals through a series of experimental studies.

First, we examine whether using a hierarchy-based metric to drive the clustering process yields solutions closer to those proposed by software developers than traditional non-hierarchical approaches. Most research on the SMC problem has focused on non-hierarchical solutions (where packages can contain only classes, not other packages) optimized using metrics such as Modularization Quality (MQ). We hypothesize that an optimization driven by a hierarchical complexity metric, specifically the Minimum Description Length (MDL) principle, will generate decompositions that better reflect the hierarchical organization patterns used by developers. A study has been conducted to test this hypothesis, and the results are presented in Chapter 3.

Second, we aim to extend the data used by the clustering algorithm beyond purely structural information by incorporating data from the software's change history. Structural dependencies — such as method calls, class imports, or attribute access — represent only one dimension of the relationships between classes. Developers often group classes not solely because they have direct structural dependencies, but because they are conceptually related and work together to implement a common feature or functionality. When developers implement a new feature or fix a bug, they typically modify all related classes together in a single transaction, known as a *commit* in version control systems. By analyzing which classes are consistently modified together over time across the project's commit history, we can infer *implicit relationships*[2] between them.

We then enhance our hierarchical clustering algorithm by enriching the dependency graph with edges derived from the project's change history. This allows the optimization process to be guided not only by the code's static structure but also by information extracted from how developers have modified related parts of the system over time. We hypothesize that this hybrid approach – combining structural

[2] Implicit relationships: relationships that exist in the developers' mental model of the system but are not captured by static structural analysis alone.

dependencies with data obtained from the project's change history – will produce hierarchical decompositions that more closely approximate the developers' reference architecture. The results of this approach are evaluated against the developers' decompositions and presented in Chapter 4.

1.4 Research questions

Following the objectives stated above, the questions that guide this research are:

- ***RQ1:*** *Does using a hierarchical-based representation and metric to guide the search for solutions to the SMC problem improve the generated solutions compared to non-hierarchical approaches?*

  Description: We aim to answer this question by comparing the results of hierarchical clustering guided by the *MDL* metric with non-hierarchical approaches guided by the *MQ* metric. In both cases, the algorithms receive only the classes of the system and the structural dependencies among them, without any information about the packages defined by the developers: the search starts from a solution in which each class is placed in its own cluster (Section 3.2), so the packages are built from scratch. We then compare the resulting decompositions with the developers' original decomposition (*DEV*), which serves as a reference for evaluation but is never provided to the algorithms. The comparison is performed across multiple dimensions. First, we examine structural characteristics of the generated decompositions, such as the number of packages, the average number of classes per package, and the maximum depth of the package hierarchy. Second, we evaluate how well the generated solutions match the organization created by the developers using similarity metrics such as MoJoFM. Third, we analyze how the different decompositions relate to the project's change history by measuring indicators such as the percentage of commits that modify classes within a single package and the average number of packages affected per commit. Through these comparisons, we seek to understand whether hierarchical clustering

can better approximate the decisions and organizational criteria that developers apply when structuring complex software systems.

- ***RQ2:*** *Does incorporating data from the software's change history to capture implicit relationships between classes improve the quality of hierarchical software decompositions?*

  Description: After establishing the benefits of hierarchical clustering in RQ1, we investigate whether enriching the optimization process with information from the project's change history produces solutions that are even more aligned with developers' architectural decisions. We introduce the Commit Relation Graph (CRG) to model relationships between classes based on how they have been modified together across commits. As in RQ1, the algorithm receives no information about the packages defined by the developers; beyond structural dependencies, it receives only relationships derived from the project's commit history. We then evaluate whether this hybrid approach—combining structural dependency information with data obtained from the project's change history—generates decompositions that more closely approximate the reference architecture than those based solely on structural dependencies. The evaluation is performed using tree edit distance metrics, which are specifically designed to compare hierarchical structures, and by examining how the generated decompositions relate to the modification records present in the project's change history.

1.5 Contributions

As part of this study, software artifacts and experimental analyses were produced. All created content is available for future research. We believe we have achieved the following contributions:

- **A hierarchical clustering algorithm guided by the MDL principle**: Co-change relationships as a relevant source of information for SMC. Empirical evidence that

the relationships between classes revealed by joint commits – that is, classes frequently modified together – complement the static structural dependencies traditionally used to address the SMC problem. When incorporated into the hierarchical MDL-based optimization, these relationships brought the generated decompositions closer to those proposed by the developers of 21 sub-modules from ten open-source systems, with statistically significant differences relative to the purely structural approach. While co-change (or logical) coupling has long been studied in software evolution research (GALL et al., 1998; ZIMMERMANN et al., 2005), this thesis provides evidence of its relevance to the search for hierarchical software decompositions;

- **A hierarchical clustering algorithm guided by the MDL principle**: Implementation and adaptation of a Hierarchical Agglomerative Clustering (HAC) algorithm that produces hierarchical decompositions of software systems, using the MDL complexity metric as the fitness function. This includes modifications to the standard HAC algorithm to enable the creation of nested package structures rather than flat partitions;

- **The Commit Relation Graph (CRG) model**: Introduction of a new representation for software projects that captures relationships between classes based on how they are modified together in the project's change history. This includes the mathematical formulation for computing relationship weights using the Jaccard similarity coefficient, as well as an efficient data structure for storing and accessing this information;

- **A hybrid hierarchical clustering approach**: Integration of the CRG model into the *MDL*-based hierarchical clustering algorithm, creating a method that considers both the structural dependencies extracted from static code analysis and the relationships inferred from how classes are modified together in the project's change history;

- **Visualization tools**: Implementation of visualization tools to graphically display hierarchical distributions produced by our optimization approach in the form of directed dependency graphs, enabling visual comparisons between different solutions and facilitating the identification of structural patterns and differences;

- **Comprehensive experimental evaluation**: Analysis and comparison of optimization results against solutions from other approaches and, most importantly, against the original decompositions created by software developers across multiple real-world software systems. This includes the collection and analysis of various metrics spanning structural characteristics, similarity measures, and indicators derived from the project's change history;

- **Public dataset and replication package**: All source code for the implemented algorithms, experimental data, including the software systems used as instances, collected metrics, and detailed instructions for replicating the studies, are publicly available in a repository to support future research and enable independent verification of the results[3].

1.6 Research methodology

The methodology adopted in this research comprises a *five-step process*. It begins with the **definition** of the research topic, problem characterization, research objectives, research questions that will be analyzed, and the methodology that will be applied in this analysis. These items are reported in this introduction and guide the development of the rest of the thesis.

After the initial definitions, we conducted a **literature review** of available approaches to software module clustering – both hierarchical and non-hierarchical – to establish the knowledge foundation needed to proceed with the research. This review prioritized publications on hierarchical clustering algorithms and their fitness functions, with particular attention to information-theoretic approaches, such as the MDL principle.

[3] Link to the repo: https://github.com/leocrantunes/hierarchical-smc

We also reviewed techniques for analyzing software repositories and extracting information from version control systems. The literature review was essential for identifying current gaps in the research area, understanding the limitations of existing approaches, and establishing the theoretical foundation for the proposed solutions. The findings from this review are presented in Chapter 2.

With the information gathered from the literature, the **solution development** was planned. This involved several key activities: (i) selecting software systems to be used as problem instances in the experiments; (ii) defining the required programming languages and libraries for implementation; (iii) developing an adapted Hierarchical Agglomerative Clustering (HAC) algorithm; (iv) implementing the MDL complexity metric for evaluating hierarchical decompositions; (v) developing mechanisms for extracting commit history data from Git repositories and constructing the Commit Relation Graph (CRG); (vi) implementing the integration between the CRG model and the hierarchical clustering algorithm; and (vii) creating the infrastructure necessary to execute the clustering algorithms systematically.

The **proposed approaches are evaluated** through controlled experiments and statistical analysis conducted in two sequential studies. In the first study, presented in Chapter 3, we compare the hierarchical MDL-based approach against a non-hierarchical MQ-based baseline across ten software systems or sub-modules extracted from larger systems. In the second study, presented in Chapter 4, we extend the evaluation to a larger set of 21 software systems or sub-modules and compare five different approaches: the non-hierarchical MQ baseline, the purely structural MDL approach (from Study 1), and three variants of the CRG-enhanced approach.

Finally, the data generated from the experiments are analyzed using descriptive statistics and correlation analysis. The results are presented through tables, graphs, and visualizations of the generated hierarchical structures. The **conclusion** summarizes findings, discusses limitations and threats to validity, and presents directions for future work.

1.7 Thesis organization

This thesis is organized into five chapters. Chapter 1 presents this introduction, including the problem definition, motivation, objectives, research questions, contributions, and methodology.

Chapter 2 presents the theoretical foundation for understanding the MDL principle, including an introduction to information theory and Kolmogorov complexity. It also reviews related work on non-hierarchical clustering approaches (focusing on local search and metaheuristics), hierarchical clustering approaches (including the foundational work by Wood and Lutz), and alternative techniques proposed for software modularization. The chapter concludes by identifying the gaps in the literature that this thesis aims to address.

Chapter 3 presents an experimental study evaluating a hierarchical clustering algorithm guided by the MDL metric. The chapter describes the adaptation of the Hierarchical Agglomerative Clustering algorithm for producing hierarchical structures, details the experimental design, the selection of software systems used as instances, and compares the results against a non-hierarchical baseline guided by the MQ metric and against the distributions proposed by software developers. Multiple metrics are analyzed, and visual comparisons of the generated structures are provided.

Chapter 4 introduces the core contribution of this thesis: the use of co-change relationships between classes, modeled by the Commit Relation Graph (CRG), to guide search-based hierarchical software clustering, and the evidence that these relationships are relevant to the SMC problem. The chapter begins by motivating the need to go beyond structural dependencies, then details the construction of the CRG from commit history data, explains its integration into the MDL-based clustering algorithm via a threshold-based approach, and presents a comprehensive quantitative evaluation demonstrating improvements over the purely structural baseline. The experimental study conducted to support the evaluation uses tree edit distance as the primary metric for comparing hierarchical structures and analyzes the results across 21 software systems.

Finally, Chapter 5 presents the conclusions, summarizes the main contributions of this research in addressing the identified research questions, discusses threats to validity and the limitations of the proposed approach, and suggests directions for future work that build upon the foundations established in this thesis.

2 BACKGROUND AND RELATED WORK

This chapter presents a theoretical foundation on information theory and a literature review on work related to non-hierarchical, hierarchical, and semantic (including LLM-based) software module clustering approaches. Finally, it states the gaps this research aims to fill.

2.1 Information Theory

Before presenting formal definitions, it is worth summarizing the intuition behind this section. The metric used in this thesis to guide the clustering process, proposed by Wood (1998), is based on the idea that a good distribution of classes into packages allows the system's dependencies to be described concisely. Imagine that we must send the dependency graph of a system to someone, bit by bit. If classes that depend on each other are grouped into the same package, most dependencies can be described by referring only to the few classes of that package, which requires short codes. If, on the contrary, dependencies are scattered across packages, many of them require longer descriptions. At the same time, each package has a cost, since its existence and contents must also be described. The Minimum Description Length (*MDL*) principle formalizes this trade-off: the best organization is the one that minimizes the total length of the description – the packages plus the dependencies described with their help. The following subsections present the concepts from Information Theory needed to compute this length.

In conversational language, we identify information about an individual object as a set of descriptions of its characteristics. We can formalize this concept computationally by defining the amount of information required to describe a finite object (such as a string) as the size of the smallest program (in bits) that, starting with a blank memory, outputs the object and then terminates. This concept of handling the quantity of information needed to

describe an individual object is called “Algorithmic Information Theory” or “Kolmogorov Complexity” (Li et al., 1990).

Kolmogorov Complexity is related to Shannon’s classical information theory (Shannon, 1948), which quantifies the information needed to transmit messages drawn from a set of possible messages with known probabilities: if all messages are equally probable, this quantity is the number of bits needed to distinguish each message from the others. Kolmogorov Complexity, in turn, applies to individual objects and does not require such probabilities. However, it cannot be used directly in practice, since the length of the smallest program that outputs a given object is not computable.

Rissanen (1978) proposed the Minimum Description Length (MDL) principle to cope with such a limitation. According to Li and Vitányi (1990), the Minimum Description Length principle states that the best theory to explain a set of data is the one that minimizes the sum of the length, in bits, of the description of the theory and the length, in bits, of the data when encoded with the help of this theory. Given some data D, the *MDL* principle states that we should pick the theory T that minimizes Equation 2.1, where $length(T)$ is the number of bits needed to minimally encode the theory T and $length(D|T)$ is the number of bits needed to minimally encode the data D given the theory T. We must be able to decode the message; otherwise, a single bit would be enough.

$$length(T) \; + \; length(D|T) \tag{2.1}$$

The Kolmogorov Complexity KC of an object x can be defined as the length of the minimum message describing x, in bits, as in Equation 2.2. In practice, it is impossible to enumerate all possible theories, so a theory to describe an object of interest must be determined around the data to be transmitted in a message.

$$KC(x) \sim \min_{i} \; length(x_i) \tag{2.2}$$

2.1.1 The MDL principle and software module clustering

While applying the MDL principle to address the SMC problem, Wood (1998) proposed a theory to describe a graph representing the design of a software system. In this graph, nodes represent the modules that convey parts of the implementation of the software system, and edges represent dependencies among these modules. Each module is represented by a unique identifier (a positive integer) and each edge is represented by a pair of positive integers representing, respectively, the identifiers of the source and target modules of the dependency.

The modularization technique proposed by Wood (1998) considers the need to communicate the graph structure between a transmitter and a receiver along a perfect transmission medium. In other words, the receiver receives exactly what is transmitted without errors, data duplication, or data loss. The receiver must be able to recreate an equivalent graph from the received message, plus know the message's structure.

In the context of this thesis, the message describes a candidate decomposition – its packages and the dependencies among its classes –, and its length in bits measures the complexity of that decomposition: the shorter the message, the simpler the decomposition.

Part of Wood's theory for describing the design of a software system requires a way of transmitting positive integers between a message's sender and receiver and those transmitted positive integers must be uniquely decodable[1]. Because the end of each integer representation cannot be detected in a sequence of bits, sending just the integer's binary representation is insufficient for decoding. Thus, the length of the binary representation is sent as extra information.

Information about the length is encoded using a prefix scheme[2] so that the end of the length field can be unambiguously detected. This is done by doubling up each digit of the length field and adding a single bit (0) at the end of the length field. This sentinel can

[1] A code is uniquely decodable if, for each source sequence of finite length, the sequence of code letters does not coincide with the sequence of code letters for any other source sequence (Wood, 1998).
[2] Prefix condition code is one in which no code word is the prefix of any other code word (Wood, 1998).

be detected since the binary representation of all positive integers starts with 1. Hence, the end of the length field is uniquely terminated by the sequence "01", which cannot occur whilst doubling up the length's binary digits.

For example, consider the transmission of the integer 27_{10}, which has a binary code of 11011_2. The length of the binary code for this number is 5_{10}, which in turn has a binary code of 101_2. Therefore, 27_{10} can be transmitted as:

$$27_{10} \cong \underbrace{11\,00\,11}_{\text{length doubled up}} \quad \underbrace{0}_{\text{extra bit}} \quad \underbrace{11011}_{\textit{integer}} \tag{2.3}$$

This concept of prefixing an integer by its length can be continually applied by prefixing the length with its length and so on until achieving a length of exactly one bit. An integer n is represented by the code sequence in Equation 2.4.

$$\mathcal{L}^* n \;=\; \log_2 n \;+\; \log_2 \log_2 n \;+\; \log_2 \log_2 \log_2 n \;+\; \ldots \tag{2.4}$$

The logarithm sequence terminates when $\log_2(\ldots \log_2 x)$ becomes less than zero. Using Kraft's Theorem (Jones, 1979, p. 48), the lower bound on the length of a code for a given integer, represented by $\log_2^* n$, is given by Equation 2.5. We can note that $\log_2^* 27 \cong 9.918$ bits compares favorably to the 12 bits observed in Equation 2.3.

$$\log_2^* n = \mathcal{L}^* n \;+\; \log_2 2.865064 \tag{2.5}$$

Such integers are needed because the message must state, for instance, how many classes, sub-packages, and dependencies each package contains. As Equation 2.4 shows, the cost of transmitting an integer grows only logarithmically with its value, but every package adds a few of these integers to the message – a fixed cost that penalizes the creation of unnecessary packages (Section 2.1.3).

2.1.2 Length of Code Words with Known Probabilities

To represent a message describing the structure of a graph, Wood proposed using the details of its edges. For a message describing a graph, this can be achieved by

determining the *a priori* probability of a node being chosen at random from the node sequence describing the graph. Let such a node be n_i and $Pr\ n_i$ be the probability of n_i being selected from the node sequence. To estimate the minimum message length for such a node sequence, the author considers that the message length of a sequence of symbols chosen from a fixed finite alphabet is equal to the sum of the corresponding length for each symbol (Equation 2.6).

$$length(s_1 \ldots s_n) = \sum_i length\ s_i \tag{2.6}$$

Hence, each symbol's length needs to be estimated. Shannon (1948) showed that the minimum description length of a symbol drawn from a finite alphabet can be estimated by $-\log_2(Pr\ s_i)$[3], where $Pr\ s_i$ is the probability of symbol s_i occurring in a message.

For instance, consider the message *aabab* consisting of two distinct symbols, a and b. The message has a total of five symbols, hence $Pr\ a = \frac{3}{5}$ and $Pr\ b = \frac{2}{5}$; since a occurs three times and b occurs twice. Hence, the minimum length for representing a is 0.74 bits, and b has a minimum length representation of 1.32 bits. Thus, the minimum length for this message is $3 \times 0.74\ +\ 2 \times 1.32 = 4.86$ bits. Observe that if we had encoded a and b using the obvious code of $a = 0$ and $b = 1$, the length would have been $5 \times 1 = 5$ bits. This length is greater than Shannon's minimum message length.

In our context, the symbols are the classes (and sub-packages) that appear as endpoints of dependencies: classes involved in many dependencies behave as frequent symbols and receive shorter codes. Moreover, the fewer the symbols among which an endpoint must be identified – for instance, the classes of a small package –, the fewer bits are needed.

To estimate the minimum message length of a graph's edge description, each edge (n_{e_1}, n_{e_2}) requires as many bits as prescribed by Equation 2.7, where n_{e_1} is the identifier of the module that is the source of the dependency represented by the edge and n_{e_2} is the module acting as target for this dependency. The negative sign on the left side of the

[3] The negative sign implies that the function decreases as the probability of the occurrence increases (Shannon, 1948).

equation is due to adding the logarithms of probabilities, which are defined in the $[0,1]$ range, and often will have negative logarithms.

$$-\sum(\log_2(Pr\ n_{e_1})\ +\ \log_2(Pr\ n_{e_2})) \tag{2.7}$$

2.1.3 Ψ: The Complexity of a Design Graph

Wood (1998) proposes that the length of a message describing all edges of a graph is a measure of the structural complexity of the graph and that a smaller message length indicates a simpler graph. Subsequently, a simpler graph represents a better design and, therefore, a better distribution of the software modules into clusters. The author proposed the message depicted in Equation 2.8 to represent a hierarchical graph.

$$\begin{aligned}
\textbf{object_description} &= \text{object_id} + \\
&\quad |nodes| + \\
&\quad |objects|\ \{\text{object}_1, \ldots, \text{object}_m\} + \\
&\quad |edges|\ \{edge_1, \ldots, edge_e\} \\
\textbf{point} &= \{object_id\} * basic_entity_id \\
\textbf{edge} &= point_1\ -\ point_2
\end{aligned} \tag{2.8}$$

In the description message above, $|x|$ encodes the cardinality of the related set x, *e.g.* $|nodes|$ represents the number of nodes contained in a particular object (or cluster). The symbols '$+$', '$-$', '$\{$', and '$\}$' are not part of the message but were added to facilitate the comprehension of the message's structure. The symbol '$+$' represents concatenation, '$*$' indicates a (possibly empty) repeating set, and '$-$' denotes a relationship.

The length of the message describing a graph representing the structure of a software's design using this encoding (Ψ) is calculated by Equation 2.9, where $\mathcal{M}$ is the set of all clusters, N_m is the number of sub-clusters and modules within cluster m, $N_m^{'}$ is the number of sub-clusters within cluster m ($N_m \geq N_m^{'}$), E_m is the number of edges described in cluster m, and $\mathcal{N}_m$ is the set of sub-clusters and modules within cluster m

($\mathcal{N}_m = |N_m|$). f_n is the frequency of module or cluster n in cluster m, which for modules is given by the degree[4] of the corresponding module, and for clusters, is the sum of the degrees of the child sub-clusters and modules plus 1 and then $F_n = \sum_n f_n$.

$$\Psi = \sum_{m \in \mathcal{M}} \left(\log_2^*(N_m + 1) + \log_2^*(N_m^{'} + 1) + \log_2^*(E_m + 1) - frequencies_m + E_m \right) + 1 \qquad (2.9)$$

$$frequencies_m = \sum_{n \in \mathcal{N}_m} f_n \times \log_2 \left(\frac{f_n}{F_n} \right) \qquad (2.10)$$

In words, Equation 2.9 adds, for each package m of the decomposition: (i) the cost of stating how many elements (classes and sub-packages), sub-packages, and dependencies m contains – the three $\log_2^*$ terms –; (ii) the cost of identifying the endpoints of the dependencies described in m, which depends on the frequencies of its classes and sub-packages (Equation 2.10); and (iii) one bit per dependency. The first part plays the role of $length(T)$ in Equation 2.1, as it describes the decomposition itself and grows with the number of packages. The other two parts play the role of $length(D|T)$, as they describe the dependencies given the decomposition, and become shorter when classes that depend on each other are grouped. Therefore, creating a package reduces Ψ only if the savings in describing the dependencies it concentrates outweigh the cost of describing the package itself.

The complexity measure was theoretically validated against Weyuker's principles (1988), which define standard properties for complexity measures. A tool has been developed on top of the metric to take an existing design and find a "simpler" alternative according to the MDL principle. The approach used a narrow beam hill-climbing driven by the defined complexity measure search strategy to look for less complex designs.

Initial empirical validation showed potential, even though have evidenced some weaknesses for a small example. One of the metric weaknesses mentioned was the lack of knowledge about the nature and use of links, e.g., not differentiating inbound from

[4] An edge, (n_i, n_j), is said *to be* incident from module n_i and incident to module n_j. The sum of the number of edges incident to and from a module is called the degree of the module (Wood, 1998).

outbound links or giving different weights for different data types. Also, the proposal did not cover a mechanism to reduce the search space and discard "unsatisfactory" changes to the design, leading to poor performance. The proposed approach improved the system's complexity by about 20 percent for a slightly more complex example.

2.2 Formal Representation of Hierarchical Decompositions

The transition from flat partitions to nested software architectures requires a fundamental shift in how software systems are represented. As presented in Section 1.2, traditional approaches to the Software Module Clustering (SMC) problem represent a system S as a directed Module Dependency Graph (MDG), denoted as $MDG = \langle N, R \rangle$, where N is the set of modules and $R \subseteq N \times N$ represents the set of structural dependencies. While the MDG is effective for non-hierarchical clustering, it is inherently limited as it cannot represent the nested organizational structures found in modern software projects.

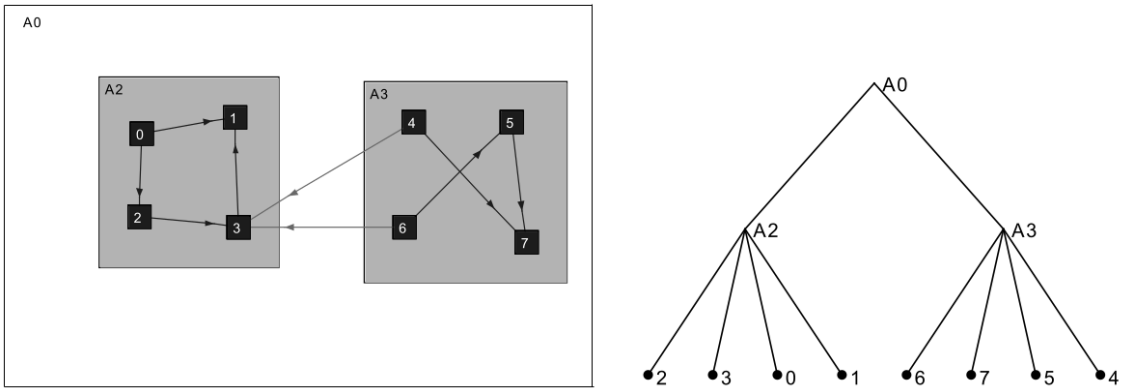


Figure 2.1: A HMD and its module tree (Lutz, 2001).

To address this limitation, we adopt the **Hierarchical Modular Decomposition (HMD)** representation, as proposed by Lutz (2001). This representation extends the formalization of a hierarchical graph by Wood (1998), as detailed in the previous section (Section 2.1.3). Figure 2.1 shows an HMD and its module tree. In this model, the system S is no longer viewed as a simple graph but as a structural pair that couples its

organization with its internal dependencies. Formally, an HMD for a system S is described as a tuple $HMD_S = \langle T_S, E_S \rangle$ where:

- T_S **(Module Tree):** An ordered and labeled tree that defines the hierarchical membership of the system.
  - The *leaf nodes* of T_S correspond to the set of modules N (e.g., Java classes);
  - The *internal nodes* of T_S correspond to the clusters or subsystems;
  - Unlike flat representations, an internal node in T_S may contain either leaf modules or other internal nodes, allowing for an arbitrary depth of hierarchy.
- E_S **(Dependencies):** A collection of directed dependencies between the modules. In the baseline structural model, E_S is equivalent to the dependency set R used in the *MDG* representation.

By adopting the HMD representation, the search algorithm explores a significantly more expansive solution space than that of non-hierarchical clustering. While traditional structural clustering focuses exclusively on partitioning modules into flat, disjoint sets, the HMD-based approach allows for the discovery and optimization of the entire parent-child relationship structure of the system. Throughout the search process, the optimizer evaluates each candidate decomposition using the complexity metric Ψ (defined in Equation 2.9) as its fitness function. The optimization objective is thus to identify the tree structure T^* that minimizes this complexity measure:

$$T^* = \arg\min_{T} \Psi(T_S, E_S) \tag{2.11}$$

For the remainder of this Thesis, an optimization driven by MDL or an MDL-based optimization means an optimization based on the MDL principle and, thus, guided by the Ψ metric.

In practice, considering packages as a hierarchy has consequences at two levels. For the SMC problem, a solution is no longer a partition but a tree, which lets us express

that two packages are more closely related to each other than to the remaining ones by placing them under a common parent package. A dependency between classes in sibling packages then crosses the boundaries of each sibling but remains internal to their parent – a distinction that flat partitions cannot represent and that Ψ takes into account, since it describes the dependencies at each level of the tree (Section 2.1.3). The evaluation is affected as well: comparing a hierarchical solution with the developers' decomposition requires metrics that consider the nesting of packages, such as the tree edit distance used in Chapter 4. For programming languages, the meaning of this nesting varies. In Java, nested packages are independent from the compiler's point of view (Section 1.2), whereas other languages attach semantics to it. In Rust, a private item may be accessed by the module where it is declared and by its descendant modules[5]; in Go, a package located under a directory named internal can only be imported by code within the tree rooted at the parent of that directory[6]; and in Python, importing a subpackage implicitly initializes its parent package, to which the subpackage is bound as an attribute[7]. Since the systems analyzed in this thesis are written in Java, the benefits of the hierarchy considered here are those related to the organization and comprehension of the system.

2.3 Non-hierarchical Software Module Clustering Techniques

Among the experiments performed on the SMC problem, those carried out by the Drexel University group stand out (Doval et al., 1999; Mancoridis; Mitchell; Rorres, et al., 1998). In several comparative experiments (Doval et al., 1999; Harman; Hierons, et al., 2002; Mitchell; Mancoridis, 2002), the results obtained by a local search algorithm using the Random Restart Hill Climbing technique were superior to those found by more complex metaheuristics, such as Genetic Algorithms and Simulated Annealing, using the same number of solution evaluations as a stopping criterion.

Based on these results, the Drexel University group developed a tool called Bunch (Mitchell, 2002; Mancoridis; Mitchell; Chen, et al., 1999) that allows selecting

[5] https://doc.rust-lang.org/reference/visibility-and-privacy.html
[6] `https://pkg.go.dev/cmd/go` (section "Internal Directories").
[7] https://docs.python.org/3/reference/import.html

between local search, a heuristic based on Simulated Annealing, and Genetic Algorithms. The Hill Climbing local search is performed for each solution from a set of randomly generated initial solutions. The number of random solutions to be generated is a configurable parameter in the tool. The neighborhood relationship used by the local search is the movement of a module from one cluster to another. Configuring the minimum number of neighboring solutions to be evaluated is also possible. For $n = 0\%$, the search proceeds from the first neighbor that represents an improved solution in comparison to the best-known solution so far; for $n = 100\%$, it examines all neighbors and proceeds from the one that yielded the best solution among them.

Another technique used to improve clustering in the Bunch tool is the identification of "omnipresent" modules. Omnipresent modules act as libraries or drivers, providing or consuming services from various subsystems in the application. The principle is that "omnipresent" modules should not be considered in clustering because they obscure the structure of the software. These modules are identified in the tool by a separate feature (*Omnipresent Module Calculator*) and separated into three categories: clients, suppliers, and clients & suppliers. A module is considered "omnipresent" if it contains three times as many dependencies as a typical module or more. After automatic identification by the tool, users can edit the list of modules, adding or removing omnipresent modules. The omnipresent modules will constitute separate subsystems (clusters) in the application.

The fitness function used in the Bunch tool is the Modularization Quality (MQ) metric, proposed by Mitchell and Mancoridis (2002; 2006; 2008). The MQ function considers a balance between cohesion and coupling, rewarding clusters containing modules having many dependencies with other modules in the same cluster and penalizing those having many dependencies with modules residing in different clusters. MQ is calculated by adding up a Cluster Factor (CF) of each cluster in the software. Below is the formula for the MQ function, where k is the number of clusters, a_i is the number of dependencies between modules within cluster C_i, and b_i is the number of dependencies that cross the boundaries of cluster C_i.

$$MQ = \sum_{i=1}^{k} CF(C_i) \tag{2.12}$$

$$CF(C_i) = \begin{cases} 0 & \text{if } a_i = 0 \\ \dfrac{a_i}{a_i + \dfrac{b_i}{2}} & \text{if } a_i > 0 \end{cases} \tag{2.13}$$

It should be noted that the same authors later modified the original MQ function (Doval et al., 1999). Mitchell *et al.* (2002) called the original MQ function "Basic MQ" and the reformulation "Turbo MQ". In later works, the authors referred to the reformulated MQ function simply as MQ. According to Mitchell, the reformulation supported weighted MDG graphs and was much faster to calculate than Basic MQ. The Turbo MQ function was used in this thesis and, as in related work on SMC, will be referred to simply as MQ. Mitchell *et al.* (2002) also proposed optimizing the calculation of the MQ function in the context of a local search in which a solution is modified by moving a single module from one cluster to another. After such a movement, instead of calculating the sum of cluster factors for all clusters, the MQ function value is updated by the difference resulting from the cluster factors of the clusters modified by the movement of the module. He called this proposal Incremental Turbo MQ (ITurboMQ).

Mitchell and Mancoridis (2001) conducted a study comparing four similarity measures. Among the measures compared, two have been used in previous studies (MoJo and precision) and two were proposed by the authors (EdgeSim and MeCl). MoJo is based on the number of cluster "Moves and Joins" required to transform a distribution of modules into clusters into another such distribution. The Precision/Recall measure (Anquetil et al., 1999) is based on pairs of clusters in the same cluster. EdgeSim uses both the position of the modules in the clusters and the dependencies between modules to calculate similarity. Merge Clusters (MeCl) calculates the distance between two distributions by dividing and joining the clusters needed to reproduce a solution. Among the results presented by Mitchell and Mancoridis (Mitchell; Mancoridis, 2001) were: i)

the four measures behaved consistently across the ten instances examined; ii) removing the omnipresent modules improved the similarity measures (on average 12.4% improvement for Precision/Recall, 9.7% for MoJo, 13.6% for EdgeSim, and 7.3% for MeCl); and iii) the EdgeSim and MeCl measures, which unlike the other approaches consider the dependencies between modules, obtained less variable results than the MoJo and Precision/Recall measures.

Wen and Tzerpos (2004) proposed the MoJoFM (MoJo eFfectiveness Measure) to correct some of MoJo's shortcomings. The shortcomings mentioned were: i) the original algorithm for calculating MoJo could be improved (Wen et al., 2003); ii) it is a non-symmetrical measure, so the number of moves and joins to transform solution A into B is different from the number to transform B into A; and iii) MoJoFM proposes a normalized and more accurate measure than MoJo. MoJoFM is calculated using the Formula 2.14, where A and B are two partitions. $mno(A,B)$ is the minimum number of moves or joins in transforming partition A into partition B, and $max(mno(A,B))$ is the maximum possible distance to transform partition A into partition B. The results indicated improved performance over MoJo and that the new metric can be used to evaluate software clustering algorithms.

$$MoJoFM(A,B) = 1 - \frac{mno(A,B)}{max(mno(A,B))} \tag{2.14}$$

Mahdavi *et al.* (2003) presented an improvement on the local search algorithm of the Bunch tool (Mancoridis; Mitchell; Chen, et al., 1999). In the initial phase, a set of executions of the local search algorithm is carried out. In this phase, a partial solution is constructed by setting the modules that share the same cluster to $\alpha\%$ of the solutions, where α is a parameter defined in the range between 10% and 100%. In the second phase, new algorithm runs are performed on a smaller search space once some modules have defined their clusters in the first phase. The results of the experimental studies indicated that more sophisticated search techniques and combinations of hybrid techniques are promising tools for tackling complex problems such as SMC.

Another work that used a local search was the comparative study conducted by Harman and Swift (2005). The authors compared the MQ fitness function with a function called EVM (Tucker et al., 2001). They evaluated both fitness functions in terms of their ability to generate consistent distributions of modules into clusters in noisy scenarios. The intention was to assess which functions can deal with dependencies that have been improperly added or not removed during software evolution. The term "noise" was used to characterize this legacy of dependencies, randomly generated in the instances before being submitted to the clustering process.

The authors concluded that the EVM function is more robust in terms of the introduction of noise in the graph representing the software, although MQ is also relatively robust. The EVM function also has the characteristic of not distinguishing between weighted and unweighted MDG graphs. The differences between the two functions (MQ and EVM) also decreased as the graphs extracted from real applications increased in size. Studies were carried out with three classes of graphs: extracted from real applications, randomly constructed, and "perfect" graphs, which were built automatically. Perfect graphs consist of graphs for which the best clustering is known and are built so that the clusters have sequentially increasing sizes, with fully connected modules within the clusters and no dependencies on modules from other clusters. Comparisons were made with perfect graphs with up to 100 modules. Noise in the graphs was simulated by mutating some of the edges. A mutation of x% in a graph means an edge has been added or removed for x% of the pairs of nodes.

Another comparative experiment involving genetic algorithms for the SMC problem was presented by Praditwong (2011), which used a crossover operator adapted for the SMC problem. The proposed adaptation combines the ideas of genetic operators used in clustering problems, such as the BPRX (Bin Packing Crossover with Replacement) proposed by Falkenauer (1996), with a new chromosome representation proposed by Harman *et al.* (2002). The results of the study indicated that the genetic algorithm using the grouping representation (Grouping Genetic Algorithm) produces better solutions than the solutions obtained with the predominantly used representation

(Grouping Number Encoding), in which each index of the chromosome represents the module and the contents represent the cluster that contains the module.

Praditwong *et al.* (2011) formulated the SMC problem as a multi-objective search problem. Instead of capturing cohesion and coupling in a single fitness function, as MQ does, their formulations represent several objectives separately. The authors proposed two formulations, the Maximizing Cluster Approach (MCA) and the Equal-size Cluster Approach (ECA), which share four objectives – cohesion (intra-cluster edges), coupling (inter-cluster edges), number of clusters, and MQ – and differ in a fifth one: MCA minimizes the number of isolated clusters, whereas ECA minimizes the difference between the sizes of the largest and the smallest clusters. Using a two-archive genetic algorithm on 17 real-world systems, they found better solutions than the single-objective approach, although at a higher computational cost. Later, Barros (2012) showed that removing composite objectives such as MQ from the ECA formulation allows a multi-objective genetic algorithm to find better solutions using less processing time.

Dias *et al.* (2003) presented seven adapted versions of the traditional genetic algorithm (Doval et al., 1999) to increase the performance and improve the quality of the solutions obtained for the problem of partitioning unweighted graphs. These adaptations were evaluated through experimental studies using artificially generated instances and the Basic MQ fitness function. The authors did not relate the studies to the SMC problem, although the underlying representation is the same.

Another comparative experiment aimed at increasing the performance of traditional genetic algorithms was presented by Botelho *et al.* (2011). The authors compared different genetic algorithms using the path reconnection technique (Glover et al., 2003), a local search procedure, and a constructive method for generating the initial population, which considers the edges' weights instead of randomly generating initial solutions. The results indicated that the adaptations using the additional local search procedure are superior to the versions using path reconnection and the constructive process.

One study that took a different approach to the evolutionary algorithms mentioned so far was carried out by Semaan *et al.* (2011). The authors proposed using the Iterated Local Search (ILS) metaheuristic to solve the SMC problem in object-oriented systems. It differs from the SMC problem presented here by treating weighted graphs and evaluating the results using the original MQ function (Doval et al., 1999).

In Semaan *et al.*'s proposal, the metaheuristic used three randomly selected perturbation procedures: (a) *random migration*, on which each module has a probability of migrating to another cluster; (b) *explosion*, on which a cluster is selected at random, and each of its modules forms a new cluster; and (c) *split*, on which a cluster is randomly selected and its modules are divided into two new clusters.

The authors used ten artificial instances with 10 to 100 vertices. The proposed ILS-based algorithm obtained better results, reaching the average fitness obtained from results in the literature (Botelho et al., 2011; Semaan et al., 2007) in less time than the best genetic algorithm of Botelho *et al.* (2011), which used the single-point crossover operator and a local search procedure.

Amoui *et al.* (2010) addressed the SMC problem by representing the reorganization of software in the form of design patterns (Gamma et al., 1995). The solution was evaluated using a normalized version of the Distance from the Main Sequence metric (Martin, 2000). This metric uses the concepts of efferent and afferent coupling. Efferent coupling is the number of classes on which a class depends. Afferent coupling is the number of classes that depend on a class. Genetic algorithms were used to find the solutions that represent the application of a sequence of design patterns. Each chromosome encodes a sequence of transformations and specific parameters according to each design pattern. Each gene in the chromosome has a variable size and comprises three types of information: a number representing the design pattern, the cluster number, and the classes in which the pattern is applied.

Two crossover operators were used: a) a single-point crossover applied to a randomly selected point at the level of the gene, changing the genes above this point, i.e., keeping the internal structure of the genes unchanged; b) a single-point crossover applied

to two genes from two different chromosomes, generating a new design pattern (gene) from the combination of the two selected genes. The mutation operator randomly selects a gene and modifies a random number of its parts. A framework (*Automatic Design Enhancer*) was developed, with two versions with different genetic operators, applied to a real project (an application called *Gold Parser*). However, no comparative results were presented about other heuristics.

Following the results of Semaan *et al.* (2011), Pinto *et al.* (2014) presented a heuristic based on *Iterated Local Search*. An extensive experiment was conducted to find the best choices for the initial solution and the perturbation components of the proposed algorithm. They also compared the ILS algorithm with various configurations of genetic algorithms. The authors showed that although the ILS metaheuristic is not widely used to solve Software Engineering problems, it proved effective for the SMC problem, outperforming the best genetic algorithm configuration in 24 of the 40 analyzed instances and using less computational processing time.

Monçores *et al.* (2018) presented a heuristic based on a constructive algorithm and the *Large Neighborhood Search* algorithm. They carried out the most extensive experimental study addressing the SMC problem at the time, involving 124 instances of varying size and complexity. They also compared the proposed algorithm with the heuristic that has found the best results for the problem to date and showed that the proposed algorithm outperformed the ILS heuristic in 93 out of 124 instances with a confidence level of 95%.

2.4 Hierarchical Software Module Clustering Techniques

Although most research work handles the SMC problem using a non-hierarchical approach, some authors considered a hierarchical structure, representing the software as a tree of nested components. Wood's work (1998), presented in Section 2.1, is an interesting example of such approaches and was highlighted for serving as the basis for this research. In the following paragraphs, we present and discuss other research addressing the SMC problem hierarchically.

Lutz (2001) proposed a variant of the metric proposed by Wood (1998) and introduced the term HMD (Hierarchical Modular Decomposition) to refer to a hierarchical structure of modules. The new metric addressed weaknesses preventing Wood's approach from generating good solutions for the examined examples. Among the changes, the author highlighted limiting the number of modules per cluster and considering the direction of the dependencies. The metric was used as the fitness function of a "steady-state" distributed genetic algorithm (Collins et al., 1991). The crossover operator was based on the tree-crossover operator (Koza, 1993) with a step for concatenating the two individuals represented by their HMD structure.

The mutation could perform three types of operations: (i) randomly move a node (a module or a cluster) to another random position in the tree without forming cycles; (ii) from a randomly selected cluster c' create a new cluster c'' made up of a subset of modules belonging to cluster c' and place this cluster within c'; and (iii) randomly select a cluster c' that is not on the first level, causing the cluster containing it to come to contain all the modules and clusters previously contained in c'. The genetic algorithm was run on three examples. The author concluded that the HMD produced by the search had its components better organized and is logically better than the organization found by Wood (Wood, 1998) for the traffic lights instance.

Even though our approach is based on both works, we extend them by using commit history to improve the results of the previous complexity metrics and by performing extensive experimentation on larger systems. We have not based our work entirely on Lutz's work because implementing the metric defined by the author did not produce the same results outlined in the article. So, we decided to take a step back and implement the original work made by Wood, which was successfully replicated.

Anquetil *et al.* (1999) evaluated the performance of four hierarchical agglomerative clustering algorithms. Hierarchical agglomerative algorithms start with individual items and gather them into small clusters according to a particular criterion. The small clusters are then successively grouped into larger clusters until a cluster containing all the items is obtained. The evaluated algorithms differ in calculating a new

cluster's distance from the others. Four different algorithms were evaluated: single linkage, complete linkage, weighted average linkage, and unweighted average linkage. The authors used the precision and recall metrics to compare different partitions of the graph of modules that represent a piece of software. Precision reports the percentage of cluster pairs found in the partitioning obtained by an algorithm in the same cluster as a reference partitioning. Recall reports the percentage of cluster pairs from a reference partitioning that are in the partitioning found by an algorithm. The authors found no significant difference in these metrics between the hierarchical and non-hierarchical algorithms. The standard partitioning provided was called an "expert partition" in the study. The expert partitions used in the study were collected from three systems (Linux, Mosaic, and a legacy system), with the modules and clusters defined according to the directory structure in which these systems were organized.

Another hierarchical clustering algorithm is ACDC (Algorithm for Comprehension-Driven Clustering), proposed by Tzerpos *et al.* (2000). Unlike algorithms based on structural information that aim to satisfy metrics such as low coupling and high cohesion, the ACDC algorithm aims for an easily understandable output, ensuring that the clusters follow familiar patterns, as well as naming the clusters intelligently. Another factor the algorithm considers is keeping the number of modules in the cluster manageable (for instance, around twenty). Experiments were carried out with two real projects. It was found that ACDC can be an effective clustering algorithm and a good candidate for reverse engineering projects. Later comparative results (Shtern et al., 2004) between the ACDC algorithm and the Bunch tool indicate that ACDC finds hierarchical partitions superior to those found by the Bunch tool. However, the latter is more effective in non-hierarchical partitions. Although ACDC considers both the semantics and the structural, it was tested in only two real-life systems, and the semantics were based on predefined patterns, such as classes residing in the same file. The approach does not consider the commit history of modules when distributing them into clusters.

Hall and McMinn (2012) analyzed the Bunch tool in the hierarchical approach. The Bunch tool was used to produce hierarchical solutions through successive clusters. A bottom-up approach was used, with each clustering grouping the clusters produced previously, i.e. the clusters acting as modules in the new clustering. Performing hierarchical clustering through successive clusters means that the different branches of the component trees have similar depths, producing a balanced tree. Another issue is that the fitness of the clusters at the last levels is altered by the fitness obtained in the clusters at the previous levels. The study results indicate an improvement of up to 30% by reducing the hierarchical levels' layering produced by the clustering. The authors presented a mutation operator that allows the formation of an unbalanced component tree. The strategy of randomly removing modules (100 times for each result produced by the tool) improved the MQ value for the five instances tested, showing that the results with unbalanced trees were superior to those with uniform branch depths.

2.5 Change History and Logical Coupling

The idea that classes modified together reveal relationships that are not visible in the source code is well established in software evolution research. Gall *et al.* (1998) introduced the notion of logical coupling by analyzing twenty releases of a large telecommunication switching system: modules that frequently changed in the same releases were considered logically coupled, even without explicit dependencies between them. Later, Gall *et al.* (2003) refined this analysis using the fine-grained transactions recorded in CVS repositories, showing that logical couplings can reveal architectural weaknesses.

Zimmermann *et al.* (2005) applied association rule mining to version histories to guide developers during maintenance: given a set of changed entities, their tool, ROSE, recommends other entities that were usually changed together with them. The strength of each rule is measured by its support – the number of transactions in which the entities changed together – and its confidence – the fraction of the transactions changing the antecedent that also changed the consequent. In an evaluation on eight open-source

projects, the three topmost suggestions contained a correct location with a likelihood of over 70%. Ying *et al.* (2004) also used association rule mining to predict source code changes, and D'Ambros *et al.* (2009) found a correlation between change coupling and software defects.

Closer to this thesis, some studies used co-change information to analyze or recover modular structures. Zimmermann *et al.* (2003) compared the coupling indicated by common changes with the coupling imposed by the system architecture, arguing that differences between both indicate anomalies that may be subject to restructuring. Silva *et al.* (2015) extracted clusters of classes that frequently change together (co-change clusters) and compared them with the package structure to assess software modularity. Beck and Diehl (2011) investigated the congruence between the package structure of 16 open-source systems and several coupling concepts, including evolutionary coupling. Later, Beck and Diehl (2013) clustered ten open-source Java systems with the Bunch tool, which optimizes MQ, using structural dependencies, evolutionary dependencies filtered by support and confidence, and their combination. Comparing the results with the package structures through MoJoFM, they found that evolutionary dependencies alone produced meaningful but worse results than structural ones, mainly because they cover fewer classes, whereas combining both sources – particularly emphasizing the dependencies present in both – improved the clustering quality. Co-change information has also been used to identify microservices in monolithic systems (Mazlami et al., 2017).

This thesis builds on these ideas but differs from previous studies in three aspects. First, co-change relationships are combined with structural dependencies in a hierarchical, MDL-based search, rather than in a flat formulation such as MQ. Second, the strength of these relationships is measured by the Jaccard coefficient and selected by percentile thresholds (Chapter 4). Third, the resulting hierarchies are compared with the developers' hierarchical decompositions using a tree edit distance, whereas Beck and Diehl (2013) evaluated flattened decompositions.

2.6 Semantic and LLM-based Approaches

Besides structural dependencies, classes may be related through the concepts they implement, which are expressed in the vocabulary of the source code – identifiers and comments. Poshyvanyk and Marcus (2006) proposed conceptual coupling metrics, which use information retrieval techniques to measure the textual similarity between classes, and showed that they capture aspects of coupling not covered by structural metrics. Kuhn *et al.* (2007) used Latent Semantic Indexing to group source code artifacts sharing a similar vocabulary, a technique they called semantic clustering. In the context of software modularization, Bavota *et al.* (2013) combined structural and semantic measures to split packages into smaller and more cohesive ones, while Bavota *et al.* (2014) used latent topics and structural dependencies to recommend moving classes between packages. Corazza *et al.* (2016) weighted the lexical information extracted from different parts of the source code – such as class names, method names, and comments – to cluster software systems for architecture recovery. More recently, Rodrigues (2025) explored word embeddings for the remodularization of Java systems.

The popularization of Large Language Models (LLMs) opened new possibilities for this kind of analysis, since these models process source code and natural language at once. Rukmono *et al.* (2024) proposed a deductive architecture recovery approach in which an LLM, guided by chain-of-thought prompting, classifies code units according to the components of a reference architecture. Zhang *et al.* (2026) combined LLMs with dependency analysis to refine the results of existing architecture recovery tools, reporting improvements in MoJoFM and other metrics on nine systems with known ground-truth architectures. LLM-based approaches are, therefore, a plausible and relatively inexpensive baseline for the SMC problem. However, evaluating them against developers' decompositions requires caution: the source code of open-source systems – including its package structure – is likely part of the data used to train these models, so an LLM may reproduce the developers' organization from memory rather than infer it (Sallou et al., 2024). Controlling this threat requires, for instance, using proprietary systems or systems whose code was not available when the model was trained.

This thesis does not use the textual content of the source code, focusing instead on structural dependencies and on the project's change history. Nevertheless, the qualitative analysis reported in Chapter 3 suggests that semantic information may complement these dimensions, as discussed in Chapter 5.

2.7 Final remarks

This chapter introduced a theoretical foundation in information theory and presented SMC-related work that used non-hierarchical, hierarchical, and semantic (including LLM-based) techniques through a literature review. The concepts underlying this research were obtained by exploratory literature research.

Most studies that tackled the SMC problem have proposed non-hierarchical techniques, meaning clusters cannot contain other clusters; they only contain modules. Local search and genetic algorithms are among the most used methods to find proper distributions of modules to clusters. In particular, local search stood out with good results yielded by algorithms such as Hill Climbing, ILS, and LNS. Even though these techniques find close to optimal solutions, their authoritativeness is very low; that is, the generated distributions were not close to those proposed by the developers. Our research aims to produce better results from this perspective.

Few studies have proposed hierarchy-based techniques. Some of these studies relied on meta-heuristics, particularly genetic algorithms. Among them, a single one has compared the proposed solutions with the developers' distributions. Besides, comparing hierarchical distributions with non-hierarchical distributions is challenging because their structures differ substantially, which also brings an opportunity to contribute to the literature.

Furthermore, although co-change information has long been used in software evolution research and has been shown to improve flat clustering when combined with structural dependencies (Beck et al., 2013), to our knowledge, no study has combined it with a hierarchical complexity metric nor evaluated the resulting hierarchies against the

developers' hierarchical decompositions. With our proposed approach, we aim to fill this gap and investigate whether it could yield better results in terms of authoritativeness.

3 COMPARING HIERARCHICAL AND FLAT CLUSTERING OF SOFTWARE MODULES

This chapter presents the first experimental study of this thesis, which compares a hierarchical, MDL-based approach to the Software Module Clustering problem with a non-hierarchical approach based on *MQ*. This chapter addresses RQ1.

3.1 Introduction

Chapter 2 has shown that several studies were carried out on hierarchical and non-hierarchical clustering approaches for software projects. However, few studies on hierarchical clustering addressed the *authoritativeness* of the distributions proposed by the algorithms, that is, how close such distributions were to those proposed by the developers. Besides, most hierarchical clustering approaches using meta-heuristics rely on genetic algorithms, while simpler local search approaches have been shown to yield good results for non-hierarchical clustering, and greedy algorithms have been shown to produce good starting solutions for these local searches.

In this chapter, we aim to complement the literature by performing experiments on hierarchical clustering algorithms using local search algorithms that depart from solutions found by greedy algorithms. We also want to compare the solutions resulting from the former with the distributions of classes to packages proposed by the developers. As discussed, research work shows that non-hierarchical clustering finds close to optimal solutions whose authoritativeness is very low. We want to examine if hierarchical clustering can lead to better results from this perspective.

This chapter is organized into six sections, starting with this introduction. Section 3.2 presents the reasoning behind the algorithm selection for this study. Section 3.3 details the criteria for selecting the software systems used in the experiments that compared the

distributions of classes to packages produced by an optimization driven by *MDL* with those produced by an optimization driven by the *MQ* metric. Section 3.4 presents the results and briefly discusses some key points the experiments revealed. Threats to validity are discussed in Section 3.5. Finally, Section 3.6 highlights some final considerations.

3.2 An algorithm for hierarchical software clustering

As described in Chapter 2, many different heuristics have been proposed to solve the SMC problem. Notably, the application of local search was shown to be promising for non-hierarchical clustering, with good results being yielded by algorithms such as the Iterated Local Search (*ILS*) (Pinto, 2014) and the Large Neighborhood Search (*LNS*) (Monçores, 2015). In both approaches, the first step of the search process involves generating a promising initial solution to ease the computational time required to execute the local search algorithm.

Pinto (2014) proposed an algorithm to find solutions for the SMC problem based on the Iterated Local Search (*ILS*) metaheuristic (Lourenço et al., 2003). *ILS* applies a local search on an initial solution. When the local search reaches a local optimum, a perturbation method is applied to its current solution, and the local search starts again from the solution generated by the perturbation. Thus, *ILS* aims to yield better solutions than those generated by random repetitions of the same local search. The metaheuristic is composed of four steps: (i) generating the initial solution, (ii) applying a local search, (iii) applying perturbation, and (iv) checking new solutions based on acceptance criteria. Algorithm 1 depicts the pseudo-code for *ILS*.

Algorithm 1: Iterated Local Search

```
1: s_0 ← generate initial solution
2: s* ← localSearch(s_0)
3: repeat
4:     s' ← perturbation(s*, history)
5:     s*' ← localSearch(s')
6:     s* ← acceptanceCriteria(s*, s*', history)
7: until stop criteria were reached
```

The first step of the search process involves generating an initial solution to ease the computational time required to execute the extended local search algorithm. Pinto's implementation of *ILS*, called *ILS_CMS*, uses a constructive approach in which the initial solution is built iteratively using a greedy algorithm (Cormack, 1971; Gordon, 1987). Greedy algorithms build up a solution in small steps, choosing a decision at each step myopically to optimize some underlying criterion. For some problems, greedy algorithms can produce a solution that is guaranteed to be close to optimal, even if it does not achieve the precise optimum (Kleinberg et al., 2006).

Pinto's greedy algorithm (Hierarchical Agglomerative Clustering - *HAC*) uses the concept of clustering of large networks (Clauset et al., 2004) to identify "community structures" – vertices more densely connected among themselves than connected to vertices in other clusters. The problem of finding "community structures" can be generically represented as identifying groups in large networks, such as the Internet, social networks, and article citation networks. The algorithm is fast enough for the hierarchical partitioning of graphs with several thousand vertices.

The first step of the algorithm is to create a solution with one module per cluster. For instance, for a problem with n modules, the initial solution has n distinct clusters, each containing a single module. Next, the algorithm checks all combinations of two clusters, selecting the union that produces the highest MQ. This step is repeated until it fails to find a union of two clusters that improves the value of MQ. The calculation of the initial solution's MQ with n clusters was implemented according to the "TurboMQ" algorithm presented by Mitchell (2002). The part of the algorithm that merges two clusters updates the current MQ value by a "delta" value, called Incremental Turbo MQ (*ITurboMQ*) in Mitchell's work. The pseudo-code for *HAC* is shown in Algorithm 2.

When moving a module from one cluster to another during the local search, it is only necessary to calculate the new *Cluster Factor* (CF) values of the two clusters involved in the movement. The current MQ value is then updated by decreasing the previous CF values of the two clusters and adding the new CFs of the same clusters. This difference avoids recalculating the CF value for all the clusters. Pinto's decision to use

Algorithm 2: Agglomerative Clustering

```
1:  s* ← generate an initial solution with one module per cluster
2:  repeat
3:    maxDelta ← 0
4:    for each c ∈ clusters do
5:      for each c' ∈ clusters do
6:        currentDelta ← calculateMergeClustersDeltaMQ(c, c')
7:        if currentDelta > maxDelta then
8:          maxDelta ← currentDelta
9:          target_c ← c
10:         target_c' ← c'
11:       end if
12:     end for
13:   end for
14:   s* ← mergeClusters(target_c, target_c')
15: until does not find clusters to merge
```

the *HAC* algorithm was made after an experimental analysis showed that its results were better than those yielded by randomly generating the initial solution for the search.

Our proposed approach is based on Pinto's work and modifies the latter by replacing *MQ* with the *MDL* fitness function to drive the optimization process using the *HAC* algorithm. Since *MDL* cannot take advantage of the *MQ* "delta" values, its value must be fully calculated in every step of the algorithm.

Alongside the new fitness function, we have modified the original algorithm to allow creating hierarchical structures. We added a step to attempt to add clusters as submodules of other clusters to see if it enhances MDL. Furthermore, at the end of the algorithm, we perform cleanup operations: (i) remove clusters without children and (ii) move clusters with just one child two levels above in the hierarchy. These modifications have been proposed after some exploratory executions. Besides, the proposal of the hierarchy movements was essential to form the hierarchical structure; otherwise, the algorithm would suggest a flat solution. The modified algorithm's pseudo-code, Hierarchical Agglomerative Clustering* (*HAC**), is presented in Algorithm 3 – the modifications were also highlighted to facilitate understanding. In the scope of this

Algorithm 3: Agglomerative Clustering*

```
1: s* ← generate an initial solution with one module per cluster
2: repeat
3:    maxDelta ← 0
4:    for each c ∈ clusters do
5:       for each c' ∈ clusters do
6:          currentDelta ← calculateMergeClustersMDL(c, c')
7:          if currentDelta > maxDelta then
8:             maxDelta ← currentDelta
9:             target_c ← c
10:            target_c' ← c'
11:         end if
12:      end for
13:   end for
14:   s* ← mergeClusters(target_c, target_c')
15: until does not find clusters to merge
16: repeat
17:   maxDelta ← 0
18:   for each c ∈ clusters do
19:      for each c' ∈ clusters do
20:         currentDelta ← calculateAddAsChildClusterMDL(c, c')
21:         if currentDelta > maxDelta then
22:            maxDelta ← currentDelta
23:            target_c ← c
24:            target_c' ← c'
25:         end if
26:      end for
27:   end for
28:   s* ← AddClusterAsChild(target_c, target_c')
29: until does not find clusters to add as children
30: s* ← removeClustersWithoutChildren(s*)
31: s* ← moveOnlyChildToGrandparent(s*)
```

document, when we mention *HAC* using *MDL* as the fitness function, we are referring to HAC^*.

3.3 Fine-tuning the proposed algorithm

Before applying the *ILS_CMS* algorithm to the *MDL* concept, we conducted a pilot study to assess how close the initial solution produced by the *HAC* algorithm was to the solution obtained from the complete *ILS* search. We evaluated whether we could use just the *HAC* algorithm in the experimental studies. Relying only on the *HAC* algorithm would reduce the processing time required to conduct the search processes, given that calculating *MDL* is much harder than calculating *MQ*, and the *ILS* algorithm demands a large number of fitness function calculations. We have selected a set of software systems to perform this study. They had to meet the following requirements:

- *Open-source software*: we chose open-source libraries due to the availability of the source code, which is necessary to execute and evaluate the proposed approach;
- *Source code is available through a Git repository*: since we needed to evaluate software evolution history, we chose systems that stored their source code in Git repositories. Git is one of the most popular version control systems in use today;
- *Source code is primarily written in Java language*: we decided to evaluate Java-based systems due to the availability of tools to extract project metadata automatically from the compiled code;
- *Different numbers of classes and packages*: to evaluate the proposed approach in different scenarios and scales, we picked systems with varying sizes, measured as the number of classes and packages (from tens to hundreds).

These requirements define eligibility but do not, by themselves, narrow the candidate set, since many Java libraries hosted on Git satisfy them. Therefore, we started from the systems used in previous studies on the SMC problem (Pinto, 2014; Monçores et al., 2018) and discarded those without a Git repository. We then supplemented this set

with systems on GitHub that met the requirements above, prioritizing the most popular ones based on indicators such as the number of stars.

Table 3.1 presents the chosen systems, their description, version, and Git repository links. We chose to work with a limited number of classes and packages for the studies reported in this chapter because of the time needed to execute *ILS_CMS* for large instances, particularly when the *MDL* fitness function was used. Therefore, we compiled each selected system and extracted sub-modules with fewer than 200 classes; systems below this limit, such as AEP Core and JavaGeom, were used as a whole. The limit was chosen as a threshold to guarantee that the optimization of each one of the instances could run in a short period (up to 30 minutes). Table 3.2 shows the sub-modules selected and their number of classes and packages.

Table 3.1: Investigated software systems

Software system	Description	Version	Repository
AEP Core	A core library providing interfaces for managing and deploying applications to Google App Engine.	0.10.0	https://github.com/GoogleCloud Platform/appengine-plugins
JavaGeom	JavaGeom is a geometry library for Java.	0.11.3	https://github.com/dlegland/ javaGeom
JUnit	The 5th major version of the programmer-friendly testing framework for Java and the JVM.	5.10.1	https://github.com/junit-team/junit5
JMetal	jMetal is a Java-based framework for multi-objective optimization with metaheuristics.	6.2.2	https://github.com/jMetal/jMetal
JGit	An implementation of the Git version control system in pure Java.	6.8.0	https://github.com/eclipse-jgit/jgit

After the software systems selection, we executed *ILS_CMS* driven by *MQ* and measured the following: (i) the percentage of time invested in the *HAC* algorithm to create the initial solution, (ii) the percentage of time invested in the remaining local search algorithm, (iii) the percentage of times *MQ* has been updated after the execution of the extended local search, and (iv) the average amount it was updated. All

Table 3.2: Investigated systems/sub-modules ordered by the number of classes.

ID	Software system	Sub-module	Classes	Packages
AEP	AEP Core	-	162	17
JGE	JavaGeom	-	159	20
JPE	JUnit	Platform Engine	135	10
JMC	JMetal	Component	132	43
JJE	JUnit	Jupiter Engine	129	9
JGP	JGit	PGM	121	5
JPL	JUnit	Platform Launcher	115	7
JGS	JGit	SSH	98	7
JMA	JMetal	Auto	69	7
JGH	JGit	HTTP	61	3

experiments were conducted on the researcher's computer configured with an Intel Core i5-1240P processor @4.40 GHz and 16 GB of RAM. All algorithms were implemented using the Java programming language SDK version 18.0.2.

Table 3.3 shows the data collected for each instance. Since the *ILS* algorithm is non-deterministic, we have run it 30 (thirty) times for each instance, and the presented results show the average for the thirty executions. We can observe that the *MQ* value did not change for most instances – such as JPE, JJE, and JGP – during the local search; that is, the solution produced by the greedy algorithm was fed into the local search algorithm, but the latter was unable to improve it.

Four instances had the *MQ* of their initial solution improved by the local search: AEP, JGE, JMC, and JMA. Although some have changed slightly (10% of the cases for JMA and 16.7% for JMC), the percentage of improvement was not more than 0.2%. For instance, in the 7^{th} run for AEP, the *MQ* value of the solution found by *HAC* was 28.0249, and after the *ILS_CMS* execution, it became 28.0684 – approximately 0.16% of improvement. Looking at the improvement percentage for all instances in Table 3.3, we can conclude that the *HAC* algorithm finds solutions that reach 99.8% of the quality of those found by the local search.

Another observation is that the percentage of time invested in the *HAC* algorithm increases as instance sizes increase. Formally, running a correlation analysis (ρ) between

Table 3.3: The distribution of the runtime for the optimization based on *MQ*, considering the percentage of time running *HAC* and *ILS_CMS*.

Instance	Classes	*HAC* (%)	*ILS_CMS* (%)	*MQ* updated (%)	Improvement (%)	Time (ms)
AEP	162	67.7	32.3	6.7	0.08	23
JGE	159	47.7	52.3	3.3	0.03	33
JPE	135	55.6	44.4	-	-	17
JMC	132	55.3	44.7	16.7	0.09	16
JJE	129	53.7	46.3	-	-	15
JGP	121	50.9	49.1	-	-	14
JPL	115	46.2	53.8	-	-	12
JGS	98	34.6	65.4	-	-	10
JMA	69	21.4	78.6	10.0	0.19	7
JGH	61	13.7	86.3	-	-	9

the number of classes and the percentage of time invested in *HAC*, we obtain a coefficient of 0.93 – indicating a strong correlation according to Cohen (1992). The greedy algorithm has a worst-case time complexity of $O(n^3)$, where n is the number of clusters. So, with larger instances, the percentage of time dedicated to the *HAC* execution tends to increase for two reasons: (i) more clusters indicate more combinations to compare on the *HAC* algorithm, and (ii) due to the high quality of *HAC*, *ILS_CMS* does not need a lot of iterations until it reaches its stop criteria.

Another advantage of *HAC* is that it is deterministic, so we avoid the multiple runs needed for non-deterministic algorithms such as *ILS_CMS*. So, even though the time to find the initial solution increases with the instance size, we save the time required for these extra executions. For instance, we ran the JGH instance 30 times, averaging 33 ms on each run; the total time was around 990 ms. 44.7% of each run was dedicated to *HAC*, that is 15.7 ms. So, for an improvement of just 0.03%, we invested 630% more time.

Due to the observed data and the advantages of *HAC*, along with simplifying the execution and better use of the available resources, we decided to use just the initial solution generated by *HAC* for the remaining studies reported here and reevaluate this decision in the future based on the results.

3.4 Analysis of different optimization drivers

We executed the *HAC* algorithm for each selected system's sub-module with both fitness functions, *MQ* and *MDL*. Subsequently, the following information has been extracted:

- *Number of packages*: the number of packages in the resulting solution;
- *Average number of classes per package*: calculated by dividing the number of classes in the project by the number of packages in the resulting solution. Nested packages are counted independently;
- *Max depth*: the longest path from the root package to the deepest package in the hierarchy. In Java, the depth of a package corresponds to the number of segments in its name and to the number of nested directories in the source tree; since nesting has no semantics in the language (Section 1.2), depth reflects only how many levels of progressively more specific groupings a developer navigates to reach a class. For the *DEV* distributions, depth was computed from the first segment of the package names (*e.g.*, *org* in $org.eclipse.jgit.http.server.glue$), whereas for the *MQ* and *MDL* distributions it was computed from the root of the generated tree;
- $MoJoFM$: calculated by comparing the optimization results with the distribution of classes to packages proposed by the developers;
- *Average number of packages affected per commit*: average number of packages with classes contained in each commit to the version control system;
- *Single-package commits ratio*: percentage of commits that contained classes belonging to a single package;
- *Execution time (in milliseconds)*: time required to run the optimization using each fitness function;
- *Graph visualization*: hierarchical visualization of the distribution of the packages in the form of a graph.

Since the *HAC* algorithm is deterministic, we have executed it once for each instance. We also compared the results with the distribution of classes into packages proposed by the developers of each project. We refer to such distribution as "*DEV*". This distribution is not provided as input to the optimization; it is only used as a reference to evaluate the resulting decompositions. The distribution resulting from the *MQ*-driven optimization was called "*MQ*", and "*MDL*" was the name given to the distribution proposed by the *MDL*-driven optimization. Next, we present the results and related discussion for each piece of information described above.

3.4.1 Number of packages

Figure 3.1 presents the number of packages for each distribution of classes into packages (*DEV*, *MQ*, and *MDL*). Nested packages were counted independently for the *DEV* and *MDL* distributions. For instance, if package A contained packages A1 and A2, the number of packages would be three: A, A1, and A2. This allows comparing the number of packages with the solution proposed by *MQ*, which contains only flat packages.

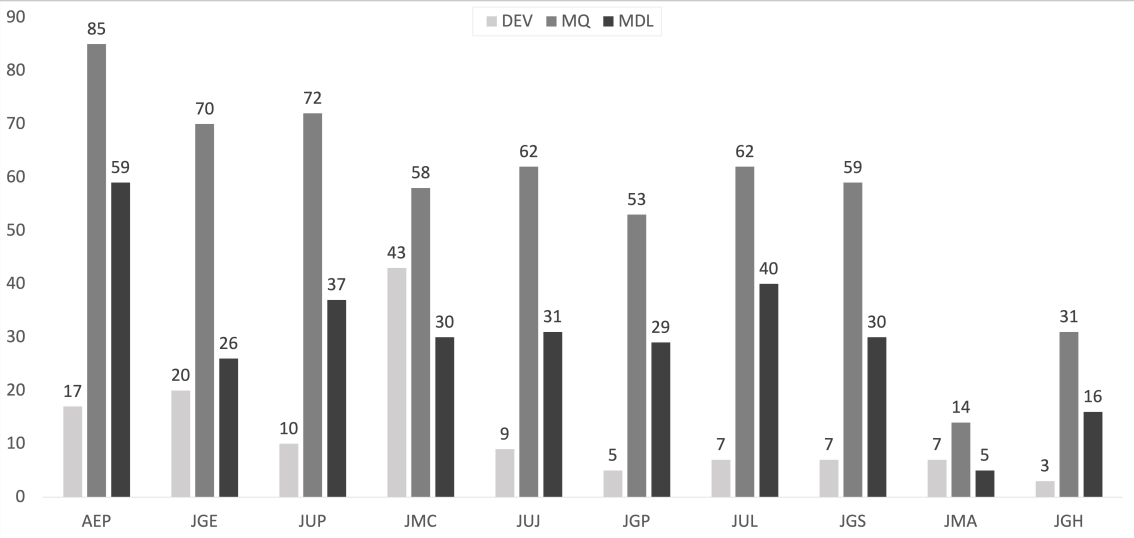


Figure 3.1: Number of packages (*DEV* x *MQ* x *MDL*).

We present the raw data in Table 3.4. We observe a large difference in the number of packages on different solutions. *MQ* generated many more packages than the *MDL*

and the *DEV* solutions for all instances, which we can observe in the MQ/DEV column and visually through the boxplot in Figure 3.2. The ratio between *MQ* and *DEV* varies from 1.35 to 10.60, with an average of 6.42 and a median of 7.04. In most cases, *MDL* generated a balanced number of packages, with a few instances producing fewer packages than *DEV* (*JMC* and *JMA*), coincidentally both sub-modules of the JMetal project. We can observe this in the MDL/DEV column. The ratio between *MDL* and *DEV* varies from 0.70 to 5.80, with an average of 3.45 and a median of 3.59.

Table 3.4: Number of packages (*DEV* x *MQ* x *MDL*).

Instance	Classes	Packages (*DEV*)	Packages (*MQ*)	Packages (*MDL*)	MQ/DEV	MDL/DEV	$MQ-MDL$
AEP	162	17	85	59	5	3.47	26
JGE	159	20	70	26	3.5	1.3	44
JPE	135	10	72	37	7.2	3.7	35
JMC	132	43	58	30	1.35	0.7	28
JJE	129	9	62	31	6.89	3.44	31
JGP	121	5	53	29	10.6	5.8	24
JPL	115	7	62	40	8.86	5.71	22
JGS	98	7	59	30	8.43	4.3	29
JMA	69	7	14	5	2	0.71	9
JGH	61	3	31	16	10.33	5.33	15
Min	-	3.0	14.0	5.0	1.35	0.70	9.0
Mean	-	12.8	56.6	30.3	6.42	3.45	26.3
Median	-	8.0	60.5	30.0	7.04	3.59	27.0
Max	-	43.0	85.0	59.0	10.60	5.80	44.0

Furthermore, the correlation coefficient (ρ) of the difference between *MQ* and *MDL* ($MQ-MDL$ column) and the number of classes is equal to 0.79, which indicates a strong correlation according to Cohen (1992). In other words, we can affirm that when the number of classes increases, the difference between the solutions proposed by *MDL* and *MQ* also increases, meaning the number of packages in *MDL* grows more slowly than *MQ*.

3.4.2 Average number of classes per package

Figure 3.3 shows each distribution's average number of classes per package (*DEV*, *MQ*, and *MDL*). We also present the raw data in Table 3.5. *MQ* has an average of

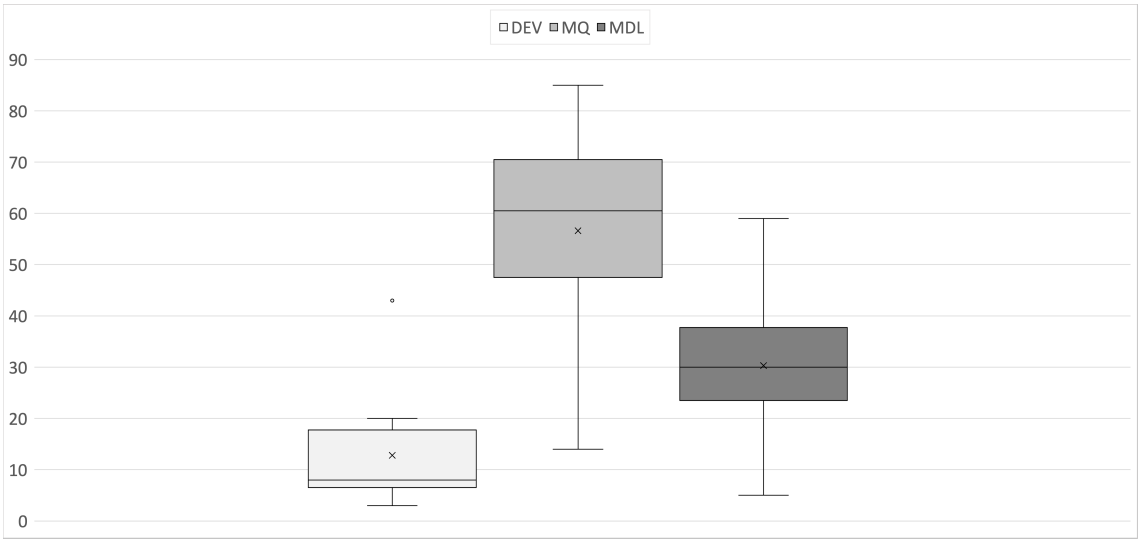


Figure 3.2: Number of packages (*DEV* x *MQ* x *MDL*) boxplot.

2.33 and a median of 2.05 classes per package, which is far different than the *DEV* distribution, which has an average of 9.14 and a median of 8.25. In contrast, *MDL* contains, on average, 5.85 classes per package with a median of 4.95 – a balanced solution compared to the previous two.

The DEV distributions show larger values for most instances, from eight to twelve classes per package on average, except for *JMC* and *JMA*, which have fewer classes per package (3.4 and 6.7, respectively) – again sub-modules of the JMetal project. This can be visualized in the boxplot in Figure 3.4. One interesting observation is that the MDL average is greater than DEV's for those instances (5.4 and 14.6, respectively).

In the MDL/DEV column, we can observe that the ratio between *MDL* and *DEV* varies from 0.31 to 2.18, with an average of 0.78 and a median of 0.5. This indicates a tendency to produce fewer classes per package than *DEV* but with a smaller variance than *MQ*, which can be observed in the MQ/DEV column. The ratio between *MQ* and *DEV* varies from 0.15 to 0.73, with an average of 0.31 and a median of 0.23.

The correlation (ρ) of the difference between *MDL* and *MQ* ($MDL - MQ$ column) and the number of classes is equal to -0.41, which indicates a medium inverted correlation according to Cohen (1992). In other words, when the number of classes

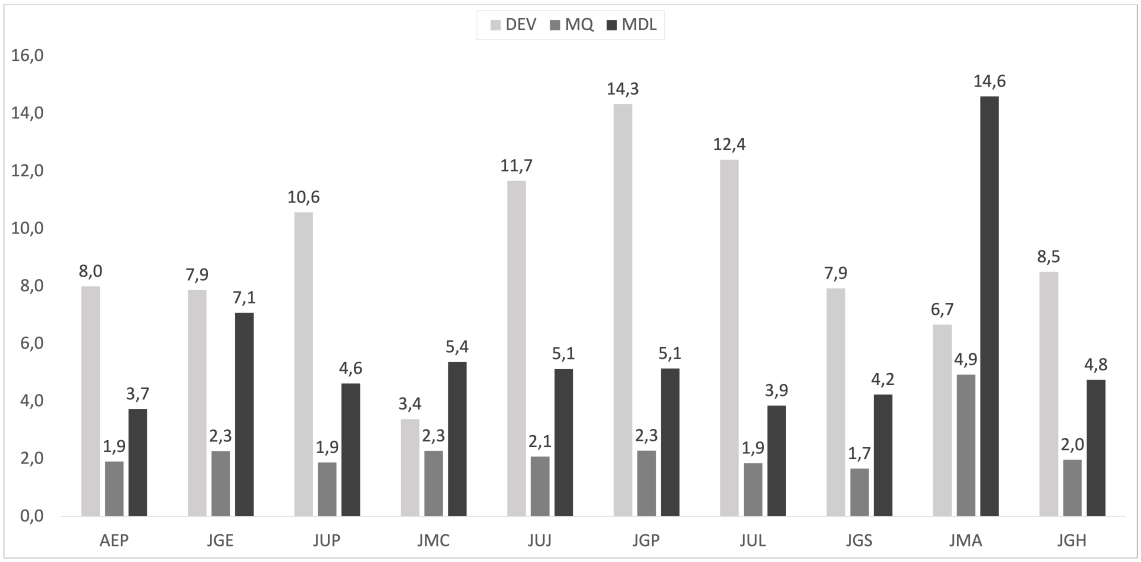


Figure 3.3: Average number of classes per package (*DEV* x *MQ* x *MDL*).

increases, the difference between the solutions proposed by *MDL* and *MQ* decreases, meaning the number of classes per package in *MDL* grows faster than *MQ*.

Table 3.5: Average number of classes per package (*DEV* x *MQ* x *MDL*).

Instance	Classes	Classes (*DEV*)	Classes (*MQ*)	Classes (*MDL*)	MQ/DEV	MDL/DEV	$MDL-MQ$
AEP	162	8	1.9	3.7	0.24	0.46	1.8
JGE	159	7.9	2.3	7.1	0.29	0.9	4.8
JPE	135	10.6	1.9	4.6	0.18	0.43	2.7
JMC	132	3.4	2.3	5.4	0.68	1.59	3.1
JJE	129	11.7	2.1	5.1	0.18	0.44	3
JGP	121	14.3	2.3	5.1	0.16	0.36	2.8
JPL	115	12.4	1.9	3.9	0.15	0.31	2
JGS	98	7.9	1.7	4.2	0.22	0.53	2.5
JMA	69	6.7	4.9	14.6	0.73	2.18	9.7
JGH	61	8.5	2	4.8	0.24	0.56	2.8
Min	-	3.40	1.70	3.70	0.15	0.31	1.80
Mean	-	9.14	2.33	5.85	0.31	0.78	3.52
Median	-	8.25	2.05	4.95	0.23	0.50	2.80
Max	-	14.30	4.90	14.60	0.73	2.18	9.70

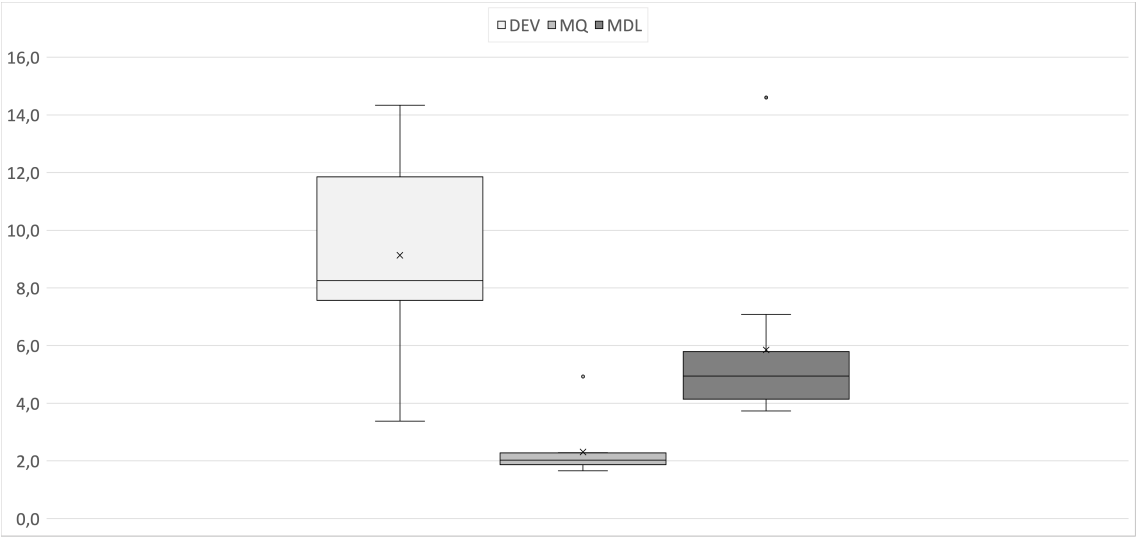


Figure 3.4: Average number of classes per package (*DEV* x *MQ* x *MDL*) boxplot.

3.4.3 Maximum depth

Figure 3.5 depicts the maximum depth of the package structure in each distribution (*DEV*, *MQ*, and *MDL*), and Table 3.6 presents the raw data. *MQ* generates flat packages, so its depth is always equal to 1. In contrast, *MDL* generates nested hierarchies, with a maximum depth varying from 2 to 10 levels (mean of 6.0), whereas the *DEV* distributions vary from 5 to 9 levels (mean of 6.4), as shown in the boxplot of Figure 3.6. However, these values are not directly comparable: the *DEV* depth includes the segments that precede the root package of each analyzed sub-module – for instance, the four levels of $org.eclipse.jgit.http$ in the JGH instance, whose classes reside in $org.eclipse.jgit.http.server$ and its sub-packages. These segments follow Java's naming convention, based on reverse domain names, rather than design decisions within the sub-module. Discounting them and counting the root package of each sub-module as the first level, the developers' hierarchies range from 2 to 6 levels (mean of 3.7), so *MDL* produced deeper hierarchies than the developers in eight of the ten instances.

The MDL/DEV column, which varies from 0.63 to 1.33, should therefore be read with this caveat in mind: the maximum depth indicates that *MDL*, unlike *MQ*, produces

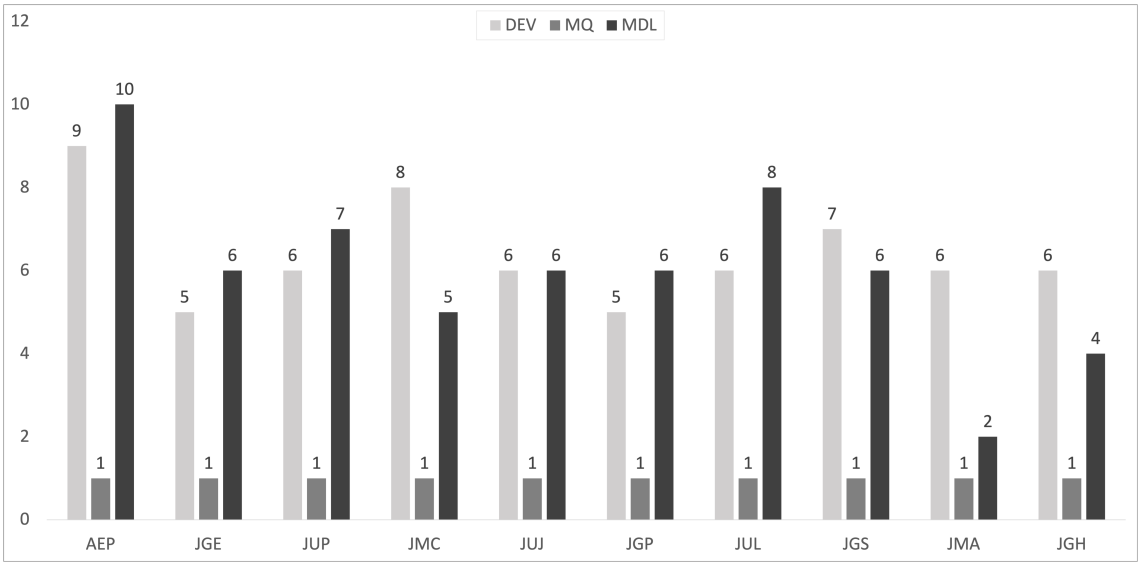


Figure 3.5: Maximum depth (*DEV* x *MQ* x *MDL*)

nested hierarchies, but not that these hierarchies have the same depth as those designed by the developers.

The correlation (ρ) of the difference between *MDL* and *MQ* (*MDL* − *MQ* column) and the number of classes is equal to 0.73, which indicates a strong correlation according to Cohen (1992). In other words, we can say that the difference between *MDL* and *MQ* grows when the number of classes increases. Since *MQ* proposes only flat solutions and is constant, we can conclude that the solutions proposed by *MDL* increase the maximum depth according to the number of classes.

3.4.4 *MojoFM*

Figure 3.7 shows *MojoFM* results for *MQ* and *MDL* compared to the *DEV* distribution. Except for *JMA*, we can observe that *MDL* values are mostly better than *MQ* values – values closer to 100 represent more similarity between the distribution yielded by the optimization process and the distribution proposed by the developers. Following the same pattern as previous metrics, the *JMA*'s *MQ* distribution *MojoFM* value is better than *MDL*.

Table 3.6: Maximum depth (DEV x MQ x MDL).

Instance	Classes	Depth (DEV)	Depth (MQ)	Depth (MDL)	MQ/DEV	MDL/DEV	$MDL-MQ$
AEP	162	9	1	10	0.11	1.11	9
JGE	159	5	1	6	0.20	1.20	5
JPE	135	6	1	7	0.17	1.17	6
JMC	132	8	1	5	0.13	0.63	4
JJE	129	6	1	6	0.17	1.00	5
JGP	121	5	1	6	0.20	1.20	5
JPL	115	6	1	8	0.17	1.33	7
JGS	98	7	1	6	0.14	0.86	5
JMA	69	6	1	2	0.17	0.33	1
JGH	61	6	1	4	0.17	0.67	3
Min	-	5.0	1	5.0	0.11	0.63	4
Mean	-	6.4	1	6.0	0.16	0.95	5
Median	-	6.0	1	6.0	0.17	1.06	5
Max	-	9.0	1	10.0	0.20	1.33	9

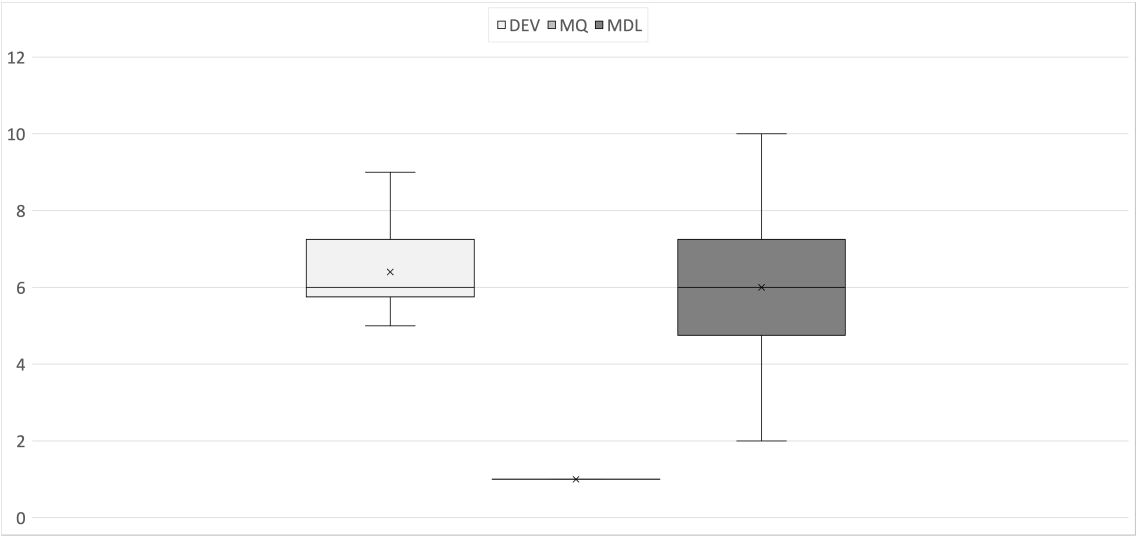


Figure 3.6: Maximum depth (DEV x MQ x MDL) boxplot.

Table 3.7 shows the $MojoFM$ values for all instances alongside relevant metrics. We can then observe that MQ values vary from 32.9 to 50.0 and have an average of 39.6 and a median of 37.6 classes per package, which is relatively different from the MDL values. MDL ranges from 35.5 to 69.0, with an average of 55.35 and a median of 58.10. The boxplot in Figure 3.8 depicts this visually.

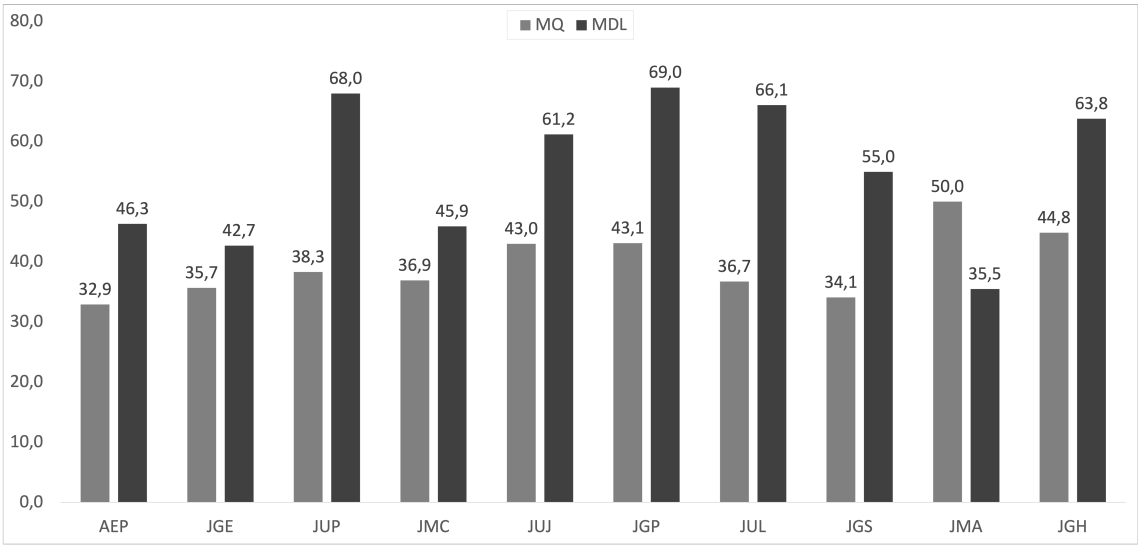


Figure 3.7: $MojoFM$ (MQ x MDL)

Table 3.7: $MojoFM$ value (MQ x MDL).

Instance	Classes	$MojoFM$ (MQ)	$MojoFM$ (MDL)	$MDL-MQ$
AEP	162	32.9	46.3	13.4
JGE	159	35.7	42.7	7.0
JPE	135	38.3	68.0	29.7
JMC	132	36.9	45.9	9.0
JJE	129	43.0	61.2	18.2
JGP	121	43.1	69.0	25.9
JPL	115	36.7	66.1	29.4
JGS	98	34.1	55.0	20.9
JMA	69	50.0	35.5	-14.5
JGH	61	44.8	63.8	19.0
Min	-	32.9	35.50	-14.5
Mean	-	39.6	55.35	15.8
Median	-	37.6	58.10	18.6
Max	-	50.0	69.00	29.7

For the *JMA* instance, the results show that the MQ distribution comes closer to the DEV distribution than the MDL proposal. Looking at the previous sections, we can infer this is due to (i) the average number of classes per package being smaller than in other instances and (ii) the generated MDL solution producing fewer packages and putting more classes in a single package. These widened the chasm between the two solutions.

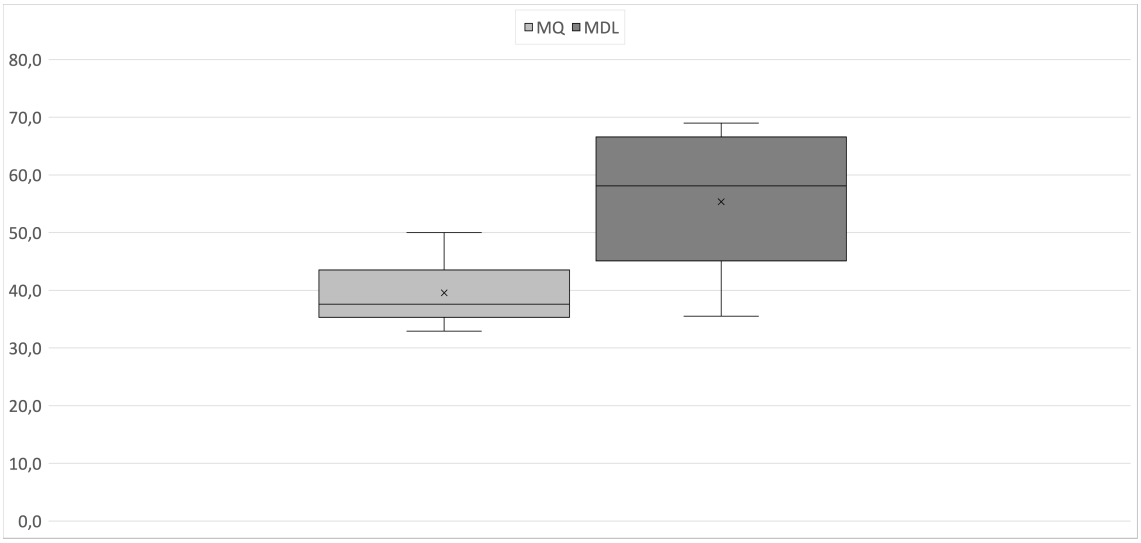


Figure 3.8: $MojoFM$ value (MQ x MDL) boxplot.

The correlation (ρ) of the difference between MDL and MQ ($MDL - MQ$ column) and the number of classes is equal to 0.22, which indicates a weak correlation according to Cohen (1992). In other words, we cannot correlate the number of classes with the difference between the solutions proposed by MDL and MQ. It might indicate that this metric varies according to other aspects of the solutions not yet considered, such as the number of dependencies.

3.4.5 Single-package commits ratio

Table 3.8 and Figure 3.9 present the percentage of commits with a single package involved. We notice better performance on the DEV distribution (more commits involving changes in classes from a single package), with MDL coming closer than MQ for all instances. For JMA, we observe that MDL has better results than the DEV distribution. This can indicate that the DEV distribution for this instance might benefit from restructuring.

In the MDL/DEV column, we observe that the ratio between MDL and DEV varies from 0.74 to 1.16, with an average of 0.91 and a median of 0.90. This indicates a tendency to produce results that are more similar to DEV than MQ regarding the

Table 3.8: Percentage of one-package commits (*DEV* x *MQ* x *MDL*).

Instance	Classes	% Commits (*DEV*)	% Commits (*MQ*)	% Commits (*MDL*)	MQ/DEV	MDL/DEV	$MDL-MQ$ (%)
AEP	162	64.7	52.3	56.9	0.81	0.88	4.59
JGE	159	40.4	23.4	33.6	0.58	0.83	10.22
JPE	135	80.7	64.1	77.1	0.79	0.96	12.95
JMC	132	67.2	54.8	64.0	0.82	0.95	9.17
JJE	129	79.9	63.6	67.1	0.8	0.84	3.49
JGP	121	86.7	79.7	84.0	0.92	0.97	4.37
JPL	115	75.4	59.7	69.7	0.79	0.93	10.00
JGS	98	50.0	39.5	44.2	0.79	0.88	4.66
JMA	69	61.1	48.7	70.6	0.80	1.16	21.9
JGH	61	84.1	57.9	61.9	0.69	0.74	3.96
Min	-	40.43	23.40	33.62	0.58	0.74	3.49
Mean	-	69.01	54.38	62.91	0.78	0.91	8.53
Median	-	71.27	56.37	65.53	0.80	0.90	6.92
Max	-	86.67	79.67	84.04	0.92	1.16	21.9

distribution of change, which can be observed in the MQ/DEV column. For *JMA*, *MDL* actually outperformed *DEV*, with 70.6% against 61.1%. The ratio between *MQ* and *DEV* varies from 0.58 to 0.92, with an average of 0.78 and a median of 0.80. This can be seen visually through the boxplot in Figure 3.10.

The correlation (ρ) of the difference between *MDL* and *MQ* ($MDL-MQ$ column) and the number of classes is equal to -0.23, which indicates a weak inverted correlation according to Cohen (1992). We wanted to determine whether the number of classes influenced the quality of the *MQ* and *MDL* solutions for single-package commits. We might say that, as the number of classes increases, the *MQ* solutions come slightly closer to the *MDL* ones.

3.4.6 Average of packages per commit

Table 3.9 and Figure 3.11 show the average number of packages per commit on each distribution (*DEV*, *MQ*, and *MDL*). This metric complements the previous metric, giving us a better view of how distant some instances are from a single-package commit, i.e., the bigger the value, the worse the performance.

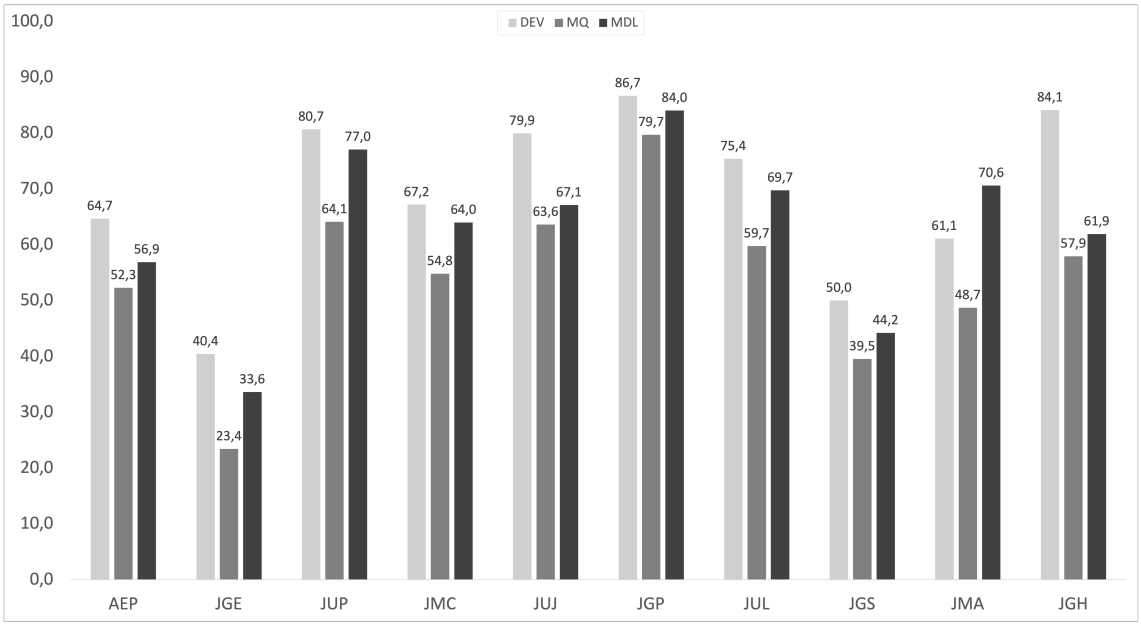


Figure 3.9: Single-package commits ratio (*DEV* x *MQ* x *MDL*)

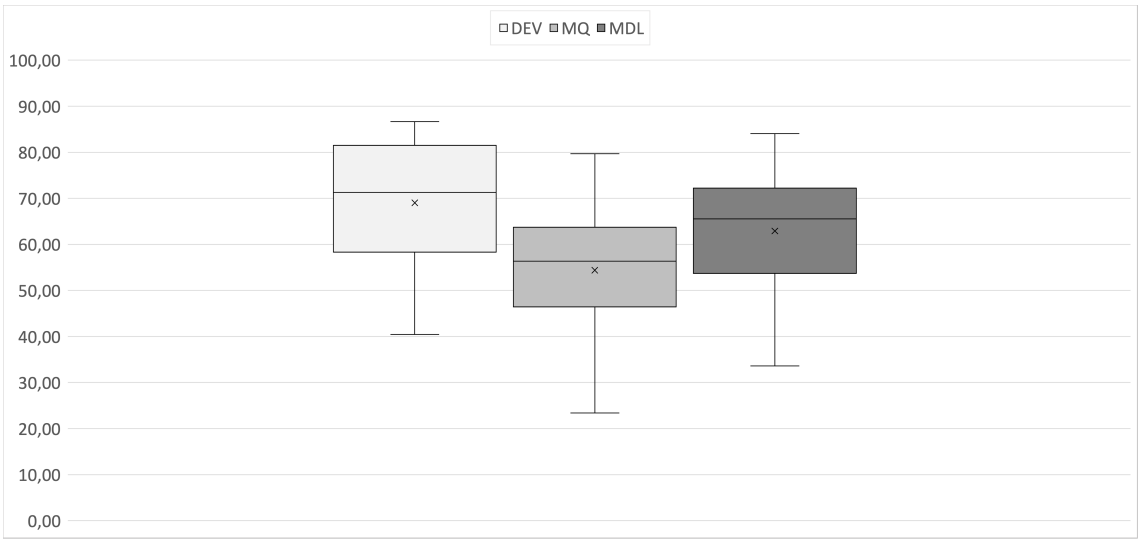


Figure 3.10: Single-package commits ratio (*DEV* x *MQ* x *MDL*) boxplot.

We can observe a very poor performance for the *MQ* distribution of *JGE* with 6.6 packages per commit on average. The remaining instances all have larger values than

Table 3.9: Average of clusters per commit (DEV x MQ x MDL).

Instance	Classes	Clusters (DEV)	Clusters (MQ)	Clusters (MDL)	MQ/DEV	MDL/DEV	$MQ-MDL$
AEP	162	1.7	2.8	2.3	1.65	1.35	0.5
JGE	159	3.0	6.6	3.1	2.20	1.03	3.5
JPE	135	1.3	2.7	1.8	2.08	1.38	0.9
JMC	132	2.0	3.0	2.1	1.50	1.05	0.9
JJE	129	1.4	2.4	1.8	1.71	1.29	0.6
JGP	121	1.2	1.7	1.4	1.42	1.17	0.3
JPL	115	1.4	2.6	2.0	1.86	1.43	0.6
JGS	98	1.7	3.0	2.5	1.76	1.47	0.5
JMA	69	1.6	2.3	1.3	1.44	0.81	1.0
JGH	61	1.2	2.7	1.9	2.25	1.58	0.8
Min	-	1.20	1.70	1.30	1.42	0.81	0.30
Mean	-	1.65	2.98	2.02	1.79	1.26	0.96
Median	-	1.50	2.70	1.95	1.74	1.32	0.70
Max	-	3.00	6.60	3.10	2.25	1.58	3.50

DEV and MDL. Once more, we observe a better performance of the MDL distribution for *JMA*, with 1.3 against 1.6 and 2.3 from DEV and MQ, respectively.

The ratio between MDL and DEV (column MDL/DEV) varies from 0.81 to 1.58, with an average of 1.26 and a median of 1.32. This indicates a tendency to produce results that are more similar to DEV than MQ, which can be observed in the MQ/DEV column. For *JMA* again, MDL actually outperformed DEV, with 1.3 against 1.6. The ratio between MQ and DEV varies from 1.42 to 2.25, with an average of 1.79 and a median of 1.74. This could also be observed from the boxplot in Figure 4.13.

The correlation (ρ) of the difference between MDL and MQ ($MDL-MQ$ column) and the number of classes is equal to 0.33, which indicates a medium correlation according to Cohen (1992). In other words, we can say that as the number of classes increases, the MQ solutions increase the average number of packages involved in a single commit with their distributions.

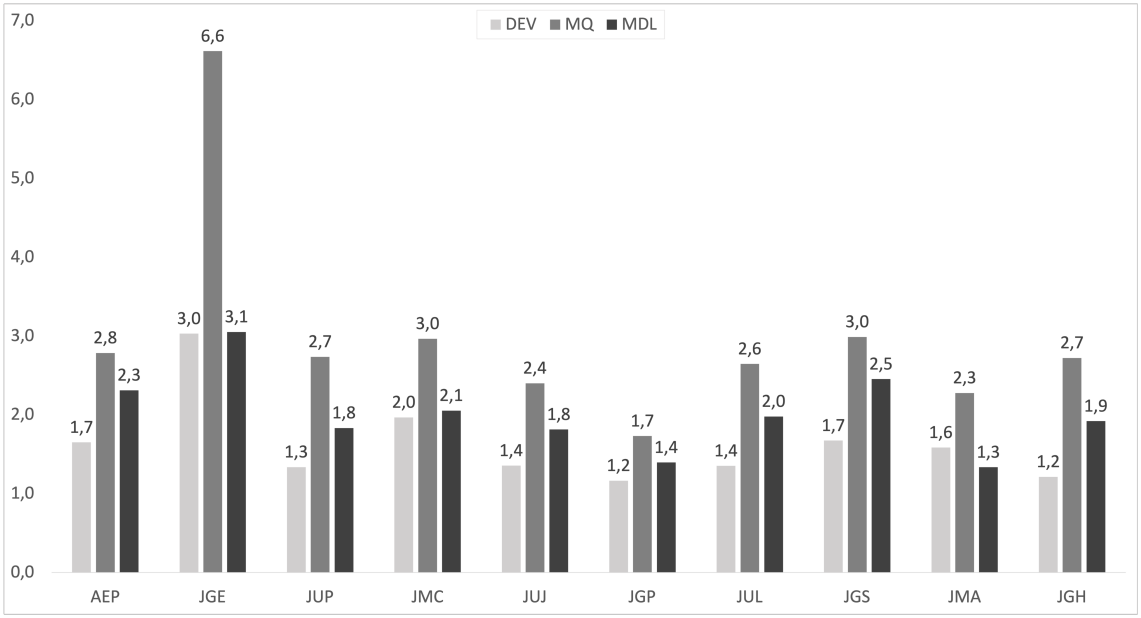


Figure 3.11: Average of packages per commit (*DEV* x *MQ* x *MDL*)

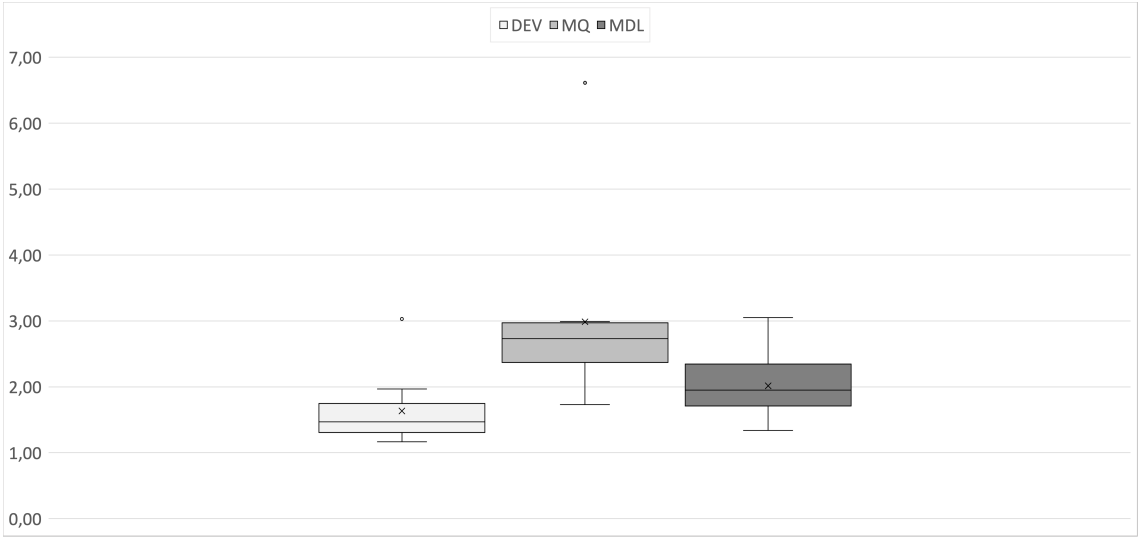


Figure 3.12: Average of packages per commit (*DEV* x *MQ* x *MDL*) boxplot.

3.4.7 Visual comparison

To show an example of the visual differences between pairs of distributions, we have plotted a directed dependency graph for JGIT's HTTP sub-module (JGH) for all three distributions: *DEV* (Figure 3.13), *MQ* (Figure 3.14) and *MDL* (Figure 3.15).

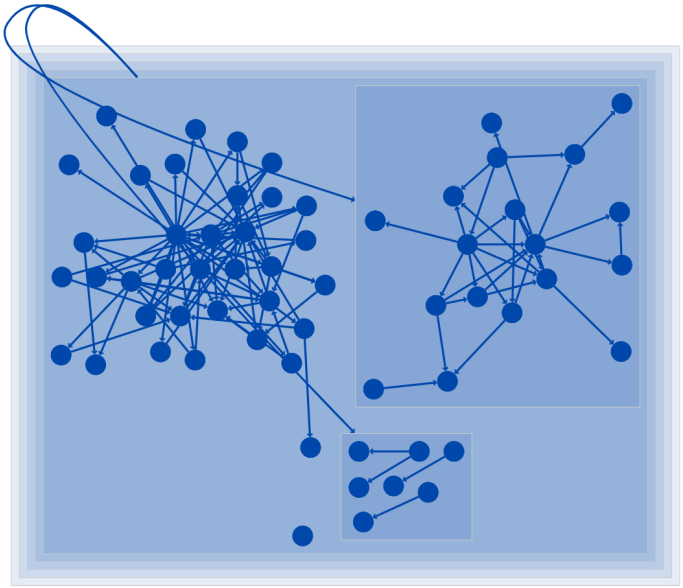


Figure 3.13: JGH *DEV* distribution.

In these plots, each circle represents a class. Arrows illustrate dependencies between classes. Each transparent rectangle wrapping a group of objects depicts a package. Objects could be either classes or inner packages. For simplicity and to facilitate the visualization, the dependencies between classes of different packages were expressed by a single arrow departing from the source class's parent package to the target class's parent package. A single arrow may describe multiple dependencies between the same packages. If two packages, A and B, depend on each other, two arrows are drawn – one from package A to B and the other from B to A. After drawing the graphs, we can observe package distribution particularities for each solution, such as (i) the number of packages and (ii) the dependencies between them.

The *MQ* distribution splits the project into many small packages, which makes it very difficult to read and understand. We can observe many packages with a single

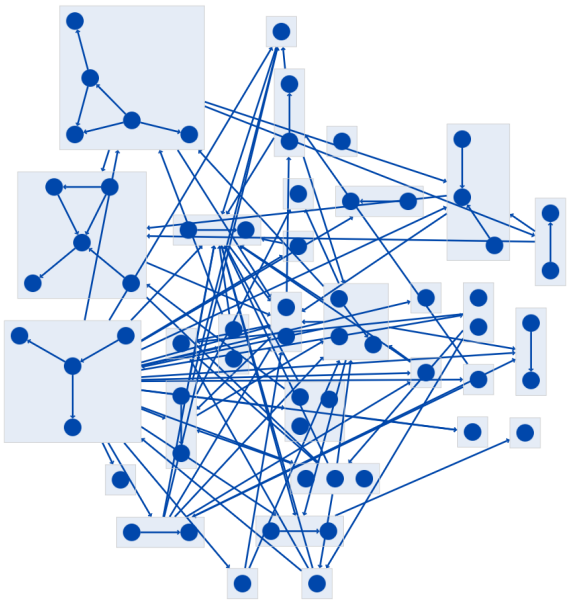


Figure 3.14: Distribution proposed by the *MQ* optimization for JGH.

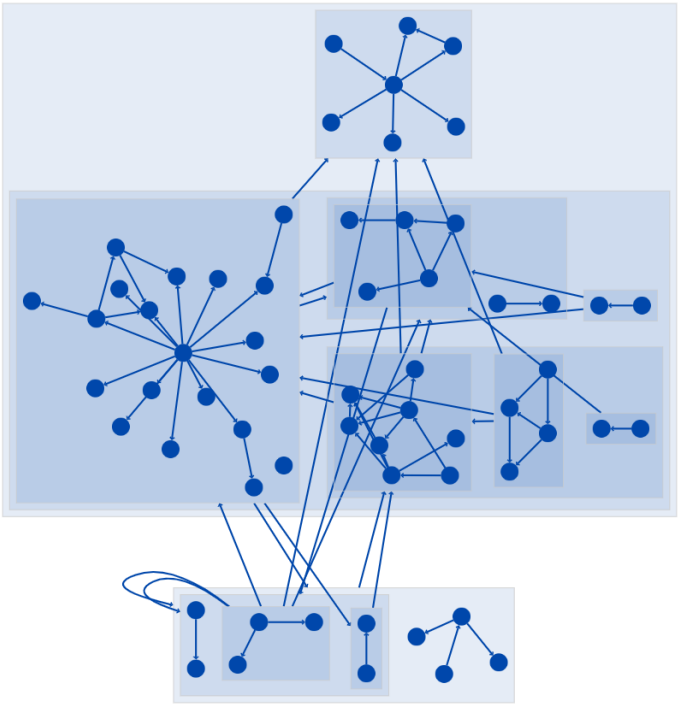


Figure 3.15: Distribution proposed by the *MDL* optimization for JGH.

class, which is also not intuitive and usual in software design. The distribution yielded by using *MDL* as a fitness function produces a balanced solution, bringing classes with high cohesion together while splitting the project into a fair number of packages and sub-packages. Compared to the *DEV* distribution, *MDL* comes visually closer but still presents significant differences in the number of inner packages.

We also observe differences among the solutions concerning the dependencies between packages. On the one hand, the distributions proposed by *MQ* have many dependencies between packages, as the number of packages is also larger. On the other hand, the *MDL* solution brings a balanced number of dependencies among packages compared to the *DEV* distribution, which has few dependencies because it does not have many packages.

Since the *JMA* instance showed poor performance for *MDL* during the studies, we investigated its generated structures and checked for the reasons that could have led to this. To help with the visualization, we have plotted its directed dependency graphs for the *DEV* (Figure 3.16) and *MDL* (Figure 3.17) distributions. We have maintained all the dependencies for these graphs, including all the cross-package dependencies (suppressed on the *JGH* example).

We noticed many dependencies among the classes in two different packages (*autoconfigurablealgorithm* and *parameter*), which could have led to package aggregation when evaluated through *MDL*. When looking at these packages, one interesting finding was that most classes inside the "parameter" package at the *DEV* distribution had the suffix "Parameter". Even though they might have dependencies on classes in other packages, they reside in the same package because of their semantics. Using *NLP* to gather semantic characteristics – such as the terms used in class names, variable names, and comments – could be beneficial. Finding a metric that considers semantics may bring interesting results, taking advantage of the *MDL* principle and its benefits to form a balanced solution.

Figure 3.18 shows the *DEV* distribution for the parameters package, and Figure 3.19 highlights how they were reorganized on the *MDL* distribution. Package names (C1,

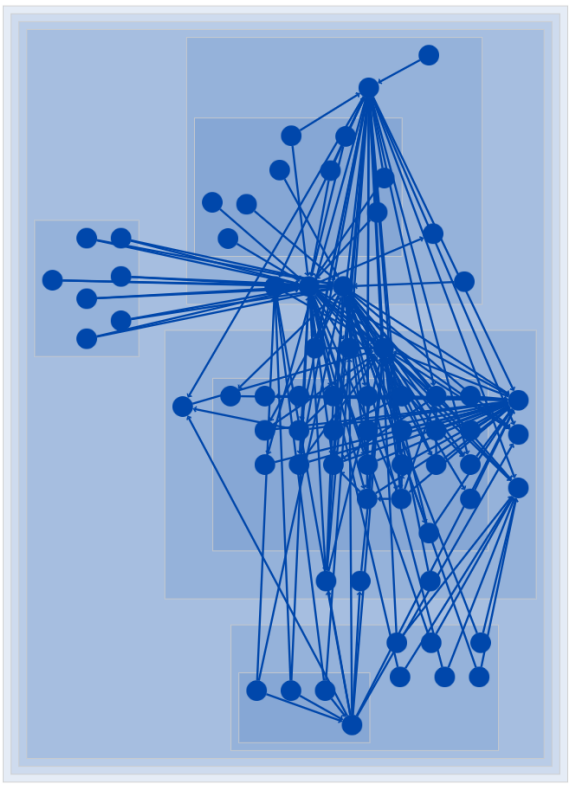


Figure 3.16: *JMA DEV* distribution.

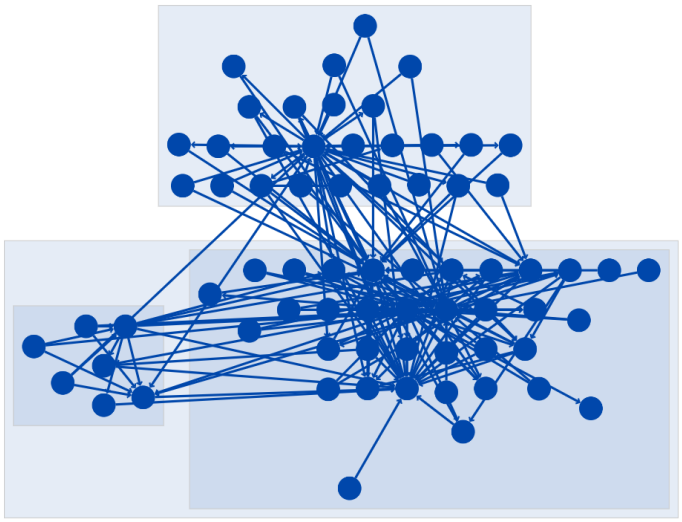


Figure 3.17: Distribution proposed by the *MDL* optimization for *JMA*.

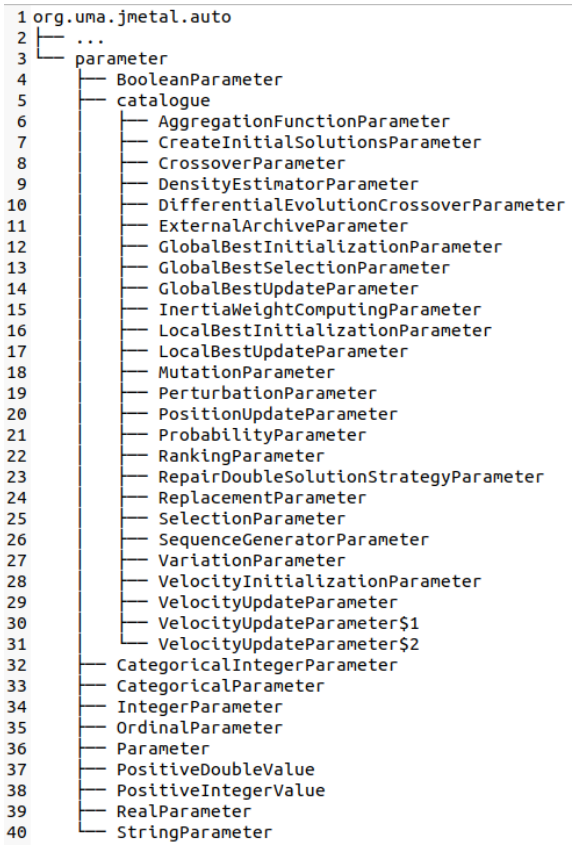


Figure 3.18: Part of the *DEV* distribution for *JMA* (parameter-related classes).

C2, and C3) are artificial. Even though this distribution has reduced the coupling and increased cohesion, it still differs from *DEV* because it did not consider the semantic characteristics of the components.

3.5 Threats to validity

According to Wohlin *et al.* (2012), four categories of validity threats may endanger the results of experimental studies: construct, internal, external, and conclusion threats.

Construct validity threats are related to generalizing the results to the concepts behind the study, in other words, the relationship between theory and observation. To minimize this kind of threat, the theory and reasoning behind the selection of the algorithm, software systems, and fitness functions were thoroughly presented in Sections

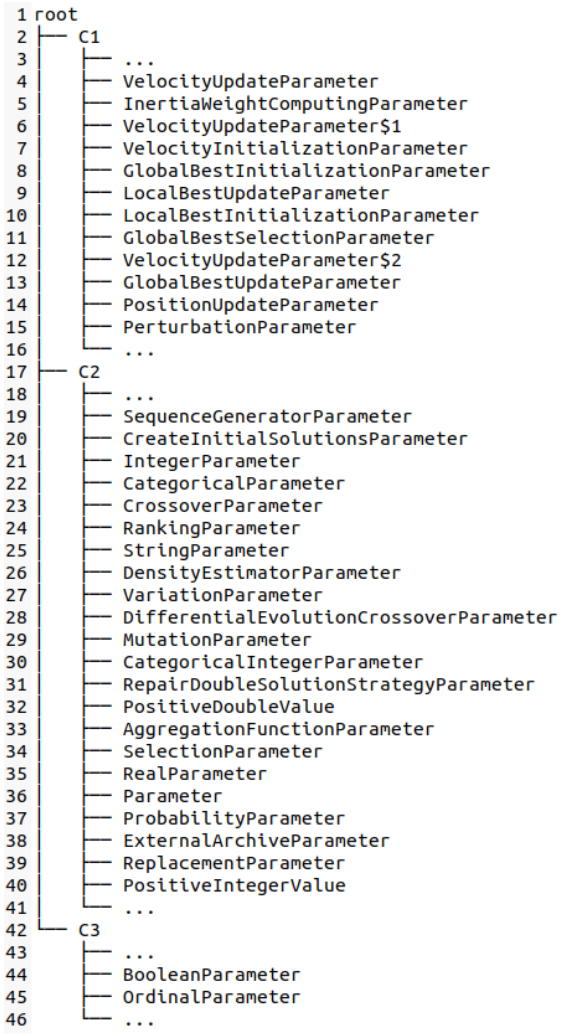


Figure 3.19: Part of the *MDL* distribution for *JMA* (parameter-related classes).

3.2 and 3.3. We have also considered open-source projects actively used and maintained in the Software Engineering community to prevent using toy examples. Moreover, most of the metrics used in this chapter describe aggregate characteristics of the decompositions, and MoJoFM – the only one that compares the assignment of classes to packages – disregards the nesting of packages. Chapter 4 addresses this limitation by adopting a tree edit distance to compare hierarchical decompositions.

The **internal validity** of experiments is threatened when the results can be tainted by modeling and measurement errors, or when we cannot replicate the behavior of the study. To guarantee the reproducibility of the experiments, all the source code and instances used in the experiments are available in a public repository[1]. Furthermore, the steps to reproduce the results are detailed in the same repository.

The **external validity** threats are related to our ability to generalize the studies' results and whether they can be used in different scopes. Only open-source software systems were explored in the experiments. This could threaten the validity of the results. Centralized development models, such as the systems used in commercial software companies, are developed with distributed and organized collaboration. In open communities, the efforts are distributed among developers around the globe with little or no centralization and control. In this case, the ideal would be to conduct a more comprehensive investigation into different commercial software systems with instances of varying sizes. However, accessing the source code of commercial software systems is difficult. In addition, the instances used in the experiments were collected from software written in Java, which can limit generalization to other programming languages. Moreover, the systems were not randomly sampled: they were drawn from previous SMC studies and complemented with popular GitHub projects, which may favor well-maintained systems.

The threats related to the inaccurate conclusions fall under the **conclusion validity** category. In this sense, only ten software systems were tested in the experiments reported in this chapter, which makes it unreasonable to adopt statistical inference tests. This limitation is addressed in Chapter 4, where the comparison between MQ and the structural MDL-based approach (HMD) is repeated on 21 sub-modules, with statistically significant differences in the normalized tree edit distance (Section 4.5.4.2).

[1] Link to repo: https://github.com/leocrantunes/hierarchical-smc

3.6 Final remarks

This chapter presented the experiments conducted on hierarchical clustering using *MDL* as a driver function to improve hierarchical structures. We have combined the *HAC* algorithm with *MDL* and compared the results with the ones generated by another driver function, MQ – widely used in the literature. The comparisons focused on measuring the authoritativeness of the approaches, comparing the solutions they proposed on an instance basis to those proposed by the developers. Overall, we noticed that *MDL* generated solutions are closer to those proposed by the developers than the ones suggested by MQ. However, there is still room for improvement.

Addressing RQ1, the experimental evidence confirms that using a hierarchical representation guided by the *MDL* metric improves the quality of the generated solutions compared to the non-hierarchical MQ baseline. This overall improvement is demonstrated across all evaluated dimensions:

- Regarding the **number of packages**, *MDL* avoided the over-fragmentation produced by MQ, resulting in a balanced package count closer to the developers' reference designs.

- In terms of the **average number of classes per package**, *MDL* produced more cohesive, better-populated groupings than the isolated, sparse packages characteristic of MQ.

- For **maximum depth**, *MDL* generated multi-level nested hierarchies, whereas MQ was restricted to a flat structure. However, after discounting the naming levels that precede each sub-module, these hierarchies were deeper than those designed by the developers in most instances.

- When evaluating authoritativeness via **MoJoFM**, *MDL* consistently outperformed MQ, indicating that it is structurally closer to the DEV distributions.

- From a change-history perspective, the **single-package commits ratio** showed that *MDL* grouped classes that are changed together more effectively than *MQ*, approaching the baseline set by developers.
- Finally, the **average number of packages affected per commit** indicated that *MDL* solutions spread the analyzed commits over fewer packages than *MQ* solutions.

Despite these improvements, the structural analysis revealed that some *MDL* solutions (such as the *JMA* instance) still exhibited discrepancies compared to the reference architecture. This divergence occurs because static structural dependencies alone cannot fully capture the implicit conceptual relationships within the developers' mental model. To bridge this remaining gap, we decided to look beyond static code analysis and examine how software actually evolves in practice. Consequently, the next chapter introduces an approach that enriches the hierarchical clustering algorithm with data extracted from the project's change history – specifically, using co-modification frequencies from version-control commits to guide the decomposition.

4 HIERARCHICAL CLUSTERING USING COMMIT DATA

This chapter presents the core contribution of this thesis: the use of co-change relationships between classes, modeled by the Commit Relation Graph (*CRG*), to guide search-based hierarchical Software Module Clustering. It details the *CRG* construction, its integration into the MDL-based clustering algorithm, and provides a quantitative and qualitative evaluation showing that these relationships bring the generated decompositions closer to those proposed by developers. This chapter addresses RQ2.

4.1 Introduction

Chapter 3 has shown that an optimization driven by *MDL* for hierarchical software clustering generates solutions that are structurally closer to developer-authored decompositions than those guided by the non-hierarchical metric *MQ*. However, there is still room for improvement. For all instances analyzed in Chapter 3, the *MDL*-based approach achieved *MoJoFM* values from 35.5 (instance JMA) to 69.0 (instance JGP). Since *MoJoFM* is a normalized metric in which 100 represents a perfect match to the authoritative decomposition, these results indicate that the solutions generated by the optimization process still required 31% to 64.5% of the maximum possible move-and-join operations to reach the developers' structure.

Because these results were obtained using only structural dependencies, they suggest that part of the remaining discrepancy may be due to information not reflected in the system's static dependency graph. In other words, the portion of the decomposition that the optimization was unable to recover may be due to complementary forms of associations between classes – beyond structural relationships – to approximate more closely the decomposition intended by the developers.

Developers often group classes that implement related features or domain concepts, even when these classes do not depend directly on each other. In such cases, the relationship between the classes is conceptual rather than structural: they share a feature, contribute to the same functionality, or belong to the same domain concept. Because this type of association does not manifest as a static dependency, it is not captured by structural analyses of the source code and, therefore, does not influence coupling or cohesion metrics derived solely from structural data.

A complementary source of information on such conceptual relationships can be found in the project's change history. When developers implement a feature or fix a defect, they typically modify all relevant classes in a single transaction (a commit). Over time, files that consistently appear together in commits tend to be associated with the same concepts or features. Thus, examining how files are modified together throughout the project's change history allows us to identify relationships among classes that are not captured by the static dependencies alone.

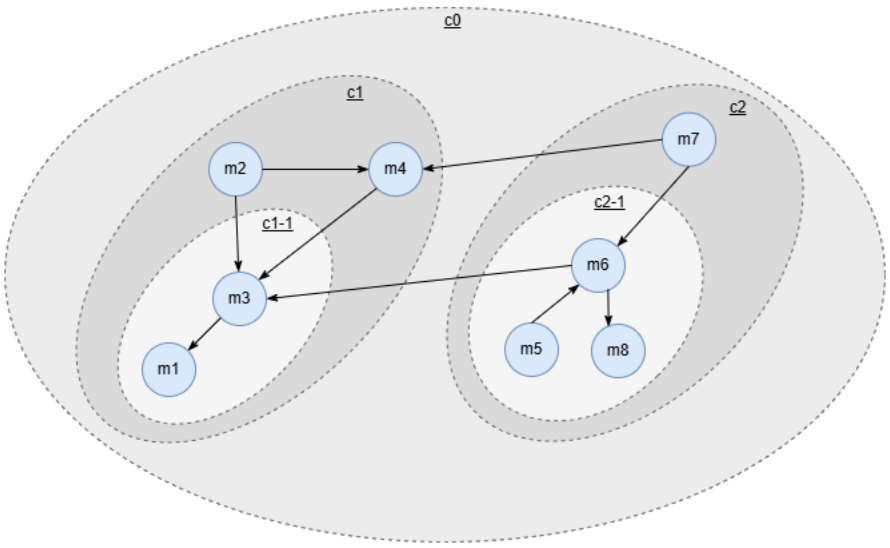


Figure 4.1: The hypothetical software system used in the following sections to illustrate the proposed approach.

These observations motivated us to extend the clustering process by incorporating information derived from the project's change history alongside the structural data. Our goal is to generate decompositions that more accurately represent the system's conceptual architecture and align more closely with the developers' intended code organization. This motivation led to the Commit Relation Graph (CRG), a representation that captures relationships among classes based on how they are modified together in commits. The following sections present the construction of the CRG, its integration into the *MDL*-based clustering algorithm, and an experimental evaluation comparing the proposed approach with a purely structural baseline.

In the following sections, we illustrate the proposed approach using an example (Figure 4.1). The hypothetical software system is comprised of eight classes ($m1$ to $m8$), organized hierarchically in five packages ($c0$, $c1$, $c2$, $c1$-1, $c2$-1). The connections between classes represent the structural dependencies among them, as in an *MDG*.

4.2 The Commit Relation Graph (CRG) Model

4.2.1 Identifying relationships from the project's change history

To capture the associations among classes described in Section 4.1, we introduce a new representation for a software project: the **Commit Relation Graph** (*CRG*). In contrast to the Module Dependency Graph (*MDG*) and the Hierarchical Module Decomposition (*HMD*) described in Chapter 2 and used in the experimental studies reported in Chapter 3, which employ directed edges to represent structural dependencies among classes, the *CRG* is a **weighted**, **complete**, and **undirected** graph. It is defined as follows:

- **Nodes** represent the classes of the system;
- **Edges** exist between every pair of nodes, indicating that a relationship between the corresponding classes can be quantified from the project's change history;

- **Edge weights** quantify the strength of the relationship between the two classes connected by the edge. The weight is a numeric value between 0 and 1, computed from the frequency with which the pair of classes is modified together in the same commits.

The graph is undirected because the relationship "modified together" is symmetric: if class A is modified together with class B, then B is likewise modified together with A. The graph is complete because it is possible to compute a numeric relationship value for any pair of classes in the system. Figure 4.2 depicts the CRG for the example system shown in Figure 4.1; edge labels display the computed weights (also seen in Table 4.1). The procedure for calculating edge weights for a real software system is described in Subsection 4.2.3.

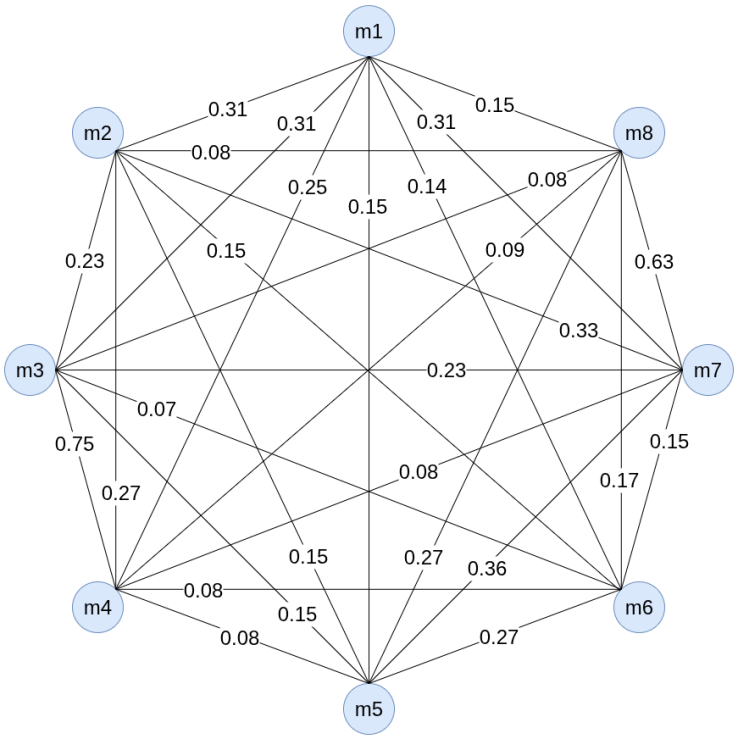


Figure 4.2: Example of a Commit Relation Graph (CRG) with eight classes.

4.2.2 Representing the Commit Relation Graph

The computational representation of the CRG requires careful consideration of memory allocation, especially for systems with a large number of classes. To store the data efficiently, we use an adjacency matrix and keep only the upper triangular portion. This is feasible because the CRG is an undirected graph, meaning that the data would be redundant above and below the matrix's main diagonal ($m1 \rightarrow m2$ is equal to $m2 \rightarrow m1$). Furthermore, we can discard the main diagonal itself, as the association of a class with itself is maximal (a class is always committed along with itself) and is represented as one. This approach reduces the memory required to store the CRG by more than half. Table 4.1 shows how the matrix looks in the example graph.

Table 4.1: Adjacency matrix of an example graph with eight classes.

	m1	m2	m3	m4	m5	m6	m7	m8
m1	–	0.31	0.31	0.25	0.15	0.14	0.31	0.15
m2	–	–	0.23	0.27	0.15	0.15	0.33	0.08
m3	–	–	–	0.75	0.15	0.07	0.23	0.08
m4	–	–	–	–	0.08	0.08	0.08	0.09
m5	–	–	–	–	–	0.27	0.36	0.27
m6	–	–	–	–	–	–	0.15	0.17
m7	–	–	–	–	–	–	–	0.63
m8	–	–	–	–	–	–	–	–

To avoid using a two-dimensional array to represent the matrix (and, consequently, unnecessary space), we define a convention and store the weights in a one-dimensional array in major-column order, i.e., column-by-column. The size of the array will always be equal to $[N \times (N-1)]/2$. To access the data, we use Formula 4.1 defined below, where M is the matrix value, i is the row, and j is the column, with $1 \leq i < j \leq N$.

$$M(i,j) = array\left[\frac{(j-1)(j-2)}{2} + (i-1)\right] \tag{4.1}$$

Table 4.2 illustrates how the data for the adjacency matrix presented in Table 4.1 is stored in a one-dimensional array. The first column in the table shows the indices in the array, the second shows the corresponding matrix indexes (row-column syntax), the third

shows the corresponding edge (with the participating classes), and the fourth presents the edge weights (relationship strength between the two classes). Also, note that the array contains 28 positions. This is because, for a CRG with eight classes, the size of the array will be equal to $[8 \times (8-1)]/2 = 28$.

To illustrate the calculation of the corresponding array index for a given element in the matrix, consider retrieving the weight of the *m*2–*m*4 edge. In this case, the row in the matrix corresponding to *m*2 (i) is 2, and the column corresponding to *m*4 (j) is 4.

Substituting these values into Formula 4.1, the corresponding array index is **4** (Formula 4.2). Therefore, the retrieved weight is **0.27**.

$$\frac{(4-1)(4-2)}{2} + (2-1) = \frac{3 \times 2}{2} + 1 = 4 \tag{4.2}$$

4.2.3 Calculating the weights of the edges

The procedure to build the CRG consists of two main steps. First, commits are preprocessed to eliminate those irrelevant to the analysis. Second, the remaining commits are analyzed to obtain the sets that modified each class pair, and the counts are normalized using a standard similarity metric to derive the edge weight.

Preprocessing of commits. Before counting occurrences, commits are preprocessed to reduce noise and to ensure that only relevant modification events are considered. The performed preprocessing includes the following operations:

- Exclude commits that do not modify source files of the classes under examination, since some commits might only touch metadata or classes related to a different sub-module;
- Resolve file renames and moves so that the history is tracked under a stable class identifier; when necessary, we map historical file paths to current class identifiers.

Counting and normalization. Let $C(m_i)$ denote the set of commits in which class m_i is modified. For any pair of classes (m_i, m_j), the number of commits in which they appear

Table 4.2: Adjacency matrix one-dimensional array for example graph with eight classes (upper triangular form).

Array index	Matrix index	Edge	Weight
0	1–2	m1–m2	0.31
1	1–3	m1–m3	0.31
2	2–3	m2–m3	0.23
3	1–4	m1–m4	0.25
4	2–4	m2–m4	0.27
5	3–4	m3–m4	0.75
6	1–5	m1–m5	0.15
7	2–5	m2–m5	0.15
8	3–5	m3–m5	0.15
9	4–5	m4–m5	0.08
10	1–6	m1–m6	0.14
11	2–6	m2–m6	0.15
12	3–6	m3–m6	0.07
13	4–6	m4–m6	0.08
14	5–6	m5–m6	0.27
15	1–7	m1–m7	0.31
16	2–7	m2–m7	0.33
17	3–7	m3–m7	0.23
18	4–7	m4–m7	0.08
19	5–7	m5–m7	0.36
20	6–7	m6–m7	0.15
21	1–8	m1–m8	0.15
22	2–8	m2–m8	0.08
23	3–8	m3–m8	0.08
24	4–8	m4–m8	0.09
25	5–8	m5–m8	0.27
26	6–8	m6–m8	0.17
27	7–8	m7–m8	0.63

together is $|C(m_i) \cap C(m_j)|$, and the number of commits in which at least one appears is $|C(m_i) \cup C(m_j)|$. We normalize these counts using the Jaccard similarity coefficient, a standard measure of set similarity. It is defined as in Formula 4.3. The Jaccard coefficient yields values in the $[0, 1]$ interval.

$$w(i,j) = \frac{|C(m_i) \cap C(m_j)|}{|C(m_i) \cup C(m_j)|} \tag{4.3}$$

A weight $w(i,j)$ close to 1 indicates that the two classes are almost always modified in the same commits (strong association). In contrast, a weight close to zero indicates classes are never or seldom modified together in the change history (weak or no association). We therefore treat $w(i,j)$ as a quantitative measure of the strength of the relationship between classes.

To illustrate the computation, Listing 4.1 shows a single commit from the project's change history. This is the initial commit of the example system, which is why all file entries are marked with the letter `A`, indicating that the files were added to the repository. The full history includes additional commits that modify subsets of these files; however, they are not shown here for brevity (see Appendix A). Each commit begins with a commit identifier (appearing in the first line), followed by metadata such as the author, date, and an optional comment.

```
commit #01
Author: Leo Antunes <leo.antunes@edu.unirio.br>
Date: Mon Mar 17 22:02:39 2025 +0000
Comment: commit inicial

A   c0/c1/c1-1/m1.java
A   c0/c1/m2.java
A   c0/c1/c1-1/m3.java
A   c0/c1/m4.java
A   c0/c2/c2-1/m5.java
A   c0/c2/c2-1/m6.java
A   c0/c2/m7.java
A   c0/c2/c2-1/m8.java
```

Listing 4.1: Example of a commit.

Based on the complete set of commits (including those not shown), Table 4.3 presents the commit sets associated with classes $m1$ and $m2$. The union of these sets contains 13 commits, and the intersection contains 4 commits. Applying Formula 4.3 yields $w(m1,m2) = 4/13 \approx 0.31$, consistent with the corresponding entry in Table 4.1.

Table 4.3: Example of Jaccard similarity calculation between two classes.

Class	**Commits**
m1.java	01, 04, 05, 07, 08, 12, 14, 15, 18
m2.java	01, 09, 11, 14, 15, 16, 18, 19
Union (total)	01, 04, 05, 07, 08, 09, 11, 12, 14, 15, 16, 18, 19
Intersection (common)	01, 14, 15, 18
Total commits in union	13
Common commits	4
Jaccard similarity	$4/13 \approx 0.31$

4.3 Integrating CRG into the MDL-based clustering algorithm

To incorporate information from the project's change history, we transform the Commit Relation Graph (CRG) into a sparse, unweighted representation that can be integrated with structural dependencies. This integration is performed through a threshold-driven selection procedure.

Given a CRG with $N \times (N-1)/2$ edge weights, we define a parameterized set of change-related edges, $E_S^{changes}$, based on a percentile $p \in [0, 100]$. For instance, a selection at the 75^{th} percentile ($p = 75$) corresponds to the top 25% strongest relationships in the *CRG*. Accordingly, the configurations HMD^{+}_{25}, HMD^{+}_{50}, and HMD^{+}_{75} evaluated in Section 4.5 include, respectively, the 75%, 50%, and 25% strongest relationships of the *CRG*. For a chosen value of p, we compute the threshold t_p of the *CRG*'s weight distribution (according to Section 4.2). Formally:

$$E_S^{changes} = \{(m_i, m_j) \in CRG \mid w(i,j) \geq t_p\}. \tag{4.4}$$

The resulting change-related edges are incorporated into the system's representation by extending the HMD definition presented in Section 2.2. We define the Enhanced HMD of a system S for a given percentile p, denoted as HMD^{+}_{p}, by redefining its dependency set as a multi-view collection of edges that includes both the structural dependencies found in the source code and the dependencies derived from the change history. Therefore, $HMD^{+}_{p} = \langle T_S, E_S \cup E_S^{changes} \rangle$, where:

- E_S represents the static dependencies extracted via source code analysis (e.g., method calls and inheritance), as in the original HMD;
- $E_S^{changes}$ represents the change-related edges selected from the CRG according to Equation 4.4, which capture implicit relationships between pairs of classes based on how frequently they are modified together in the project's version control history.

By evaluating HMD_p^+, the Ψ fitness function takes into account both the structural dependencies and the change history of the software during the optimization process.

Figure 4.3 illustrates the HMD_{75}^+ for our example system, which includes only change-related edges with Jaccard weights of at least **0.27**. In this figure, dashed edges represent relationships derived from the *CRG*, while solid edges represent original structural dependencies.

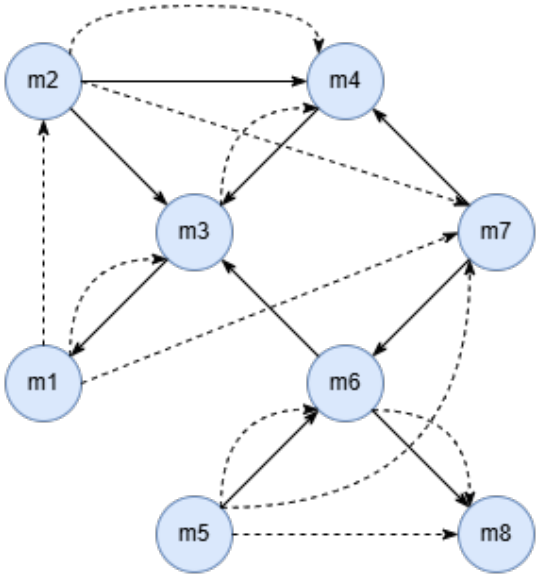


Figure 4.3: Example of an enhanced *HMD*.

The construction of the HMD_p^+ follows three fundamental integration rules:

- *Single dependency mapping:* Although CRG relationships are conceptually symmetric, they are incorporated into the HMD_p^+ as single directed edges

$(m_i \rightarrow m_j)$ where $i < j$. This approach was taken to avoid over-weighting the graph with two distinct edges, since one connection would be enough for strengthening the relationship among the classes;

- *Multi-view graph dependencies:* The final dependency set in the HMD_p^+ is the union of static structural edges (E_S) and the set of threshold-bounded, change-related edges ($E_S^{changes}$), and they are treated equally by the algorithm;

- *Reinforcement of existing dependencies:* If a structural dependency is already present for a given pair of classes, the CRG-derived edge is likewise added. This intentional duplication strengthens the relationship between the classes in the enhanced graph, reflecting both structural interaction and evidence from the project's change history.

This mechanism impacts the Ψ complexity metric. In the *MDL* formulation (Equation 2.9), the description length is calculated based on the frequency of interactions (f_n) of each class. By adding *CRG* edges alongside structural ones, we effectively increase the "information density" between specific classes.

For the HAC^* algorithm, which seeks to minimize Ψ at each merge step, these reinforced links act as attractors. Mathematically, the bit cost[1] of separating classes with reinforced links across different packages is significantly higher than separating classes connected by only a single source. This occurs because the *MDL* principle uses optimal prefix codes, where high-frequency interactions (those with higher f_n) are assigned shorter bit-string representations to minimize the overall description. Forcing such frequent interactions into cross-package dependencies requires a more verbose encoding to identify the target modules across different packages, thereby increasing the total number of bits required to describe the system. Consequently, because the HAC^* algorithm is greedy, it prioritizes merging these high-frequency pairs early in the process to achieve the largest possible reduction in the total description length.

[1] bit cost is defined as the specific contribution of an architectural decision to the total message length Ψ

To illustrate the threshold mechanism, Table 4.4 presents the edge weights of the example system sorted in ascending order. The threshold values for CRG_{25}, CRG_{50}, and CRG_{75} – corresponding to the 25^{th}, 50^{th}, and 75^{th} percentiles – are 0.09, 0.15, and 0.27, respectively. These values determine which edges are included in each HMD_p^+ configuration.

Table 4.4: Jaccard similarity values between classes, grouped by CRG.

Source	**Target**	**Jaccard**	*CRG*	CRG_{25}	CRG_{50}	CRG_{75}
m3	m6	0.07	✓			
m2	m8	0.08	✓			
m3	m8	0.08	✓			
m4	m7	0.08	✓			
m4	m5	0.08	✓			
m4	m6	0.08	✓			
m4	m8	0.09	✓	✓		
m1	m6	0.14	✓	✓		
m1	m5	0.15	✓	✓		
m1	m8	0.15	✓	✓		
m2	m5	0.15	✓	✓		
m2	m6	0.15	✓	✓		
m3	m5	0.15	✓	✓		
m6	m7	0.15	✓	✓	✓	
m6	m8	0.17	✓	✓	✓	
m2	m3	0.23	✓	✓	✓	
m3	m7	0.23	✓	✓	✓	
m1	m4	0.25	✓	✓	✓	
m2	m4	0.27	✓	✓	✓	
m5	m6	0.27	✓	✓	✓	
m5	m8	0.27	✓	✓	✓	✓
m1	m2	0.31	✓	✓	✓	✓
m1	m3	0.31	✓	✓	✓	✓
m1	m7	0.31	✓	✓	✓	✓
m2	m7	0.33	✓	✓	✓	✓
m5	m7	0.36	✓	✓	✓	✓
m6	m8	0.63	✓	✓	✓	✓
m3	m4	0.75	✓	✓	✓	✓

4.4 Optimization over the HMD_p^+ configuration

Once the dependency set $E_S \cup E_S^{changes}$ is established for a specific percentile configuration p, the search for the optimal hierarchical structure is performed. As in the purely structural case (Equation 2.11), the objective is to find the tree structure T^* that minimizes the complexity of the enhanced dependency set:

$$T^* = \arg\min_T \Psi(T_S, E_S \cup E_S^{changes}) \tag{4.5}$$

This process uses the MDL-based Hierarchical Agglomerative Clustering (HAC^*) algorithm, as introduced in Section 3.2, now operating over the multi-view dependencies of the Enhanced HMD.

The optimization is driven by minimizing the complexity measure Ψ. During each iteration, the algorithm evaluates all possible merges between packages and calculates the resulting description length in bits. In the HMD_p^+ context, the fitness function treats structural and version control dependencies as distinct edges in the same graph.

The presence of reinforced edges – where a pair of classes share both static and CRG-derived links – has a decisive impact on the greedy behavior of HAC^*. Because Ψ is sensitive to the frequency of interactions within a package (f_n in Equation 2.10), reinforced links act as high-frequency signals. These signals create a significant penalty, measured in bits in the message, for separation; consequently, the optimizer is biased toward grouping these high-confidence pairs early in the clustering process.

After the agglomerative phase reaches a local minimum, the algorithm performs the HAC^* refinement steps – handling child-package additions and structural cleanup – to ensure the resulting T^* is an optimized, non-flat hierarchy.

4.5 Quantitative evaluation of the approach

We conducted an experimental study to evaluate whether incorporating commit history through the HMD_p^+ representation improves hierarchical clustering of software modules compared to using structural dependencies alone. This evaluation aims to

determine whether data from change history effectively bridges the gap identified in the purely structural *MDL* approach. The study is organized into four subsections: research question and metrics (4.5.1), an illustrative example on how the metrics are applied (4.5.2), the instance selection process (4.5.3), and a comparative analysis using the predefined metrics (4.5.4).

4.5.1 Research Questions and Metrics

This experimental study addresses **RQ2**: *Does incorporating data from the software's change history to capture implicit relationships between classes improve the quality of hierarchical software decompositions?*

To answer this question, we evaluate whether incorporating change history information through HMD_p^+ representations produces hierarchical decompositions that are closer to the developer-authored (DEV) distributions than those produced by structural dependencies alone. We assess the generated solutions from two complementary perspectives, using the metrics described below. The first is **authoritativeness** – the degree to which a decomposition aligns with the developers' intended organization – measured by the number of packages (NP) and the tree edit distance (ZSS) to DEV. As usual in the SMC literature (Anquetil et al., 1999; Wen et al., 2004; Barros et al., 2015), we adopt the developers' decomposition as the reference. We do not assume that DEV is optimal; rather, it is the organization adopted by the developers who maintain each system and, therefore, the best available proxy for the organization this thesis aims to approximate. The second perspective is change localization, measured by SPCR and APC. For these metrics, DEV serves as a baseline rather than as a target: a decomposition may localize past changes better than the developers' one, as observed in Chapter 3 for the JMA instance, whose MDL solution – obtained without any change-history data – outperformed DEV on both metrics.

- *Number of packages (NP):* This metric counts the total number of packages in the resulting solution. To accurately reflect the depth of hierarchical decompositions, nested packages are counted independently. For instance, if package A contained

packages A1 and A2, the number of packages would be three: A, A1, and A2. This allows for a fair comparison between the flat packages produced by *MQ* and the nested structures produced by *MDL* and *DEV*;

- *Tree Edit Distance (ZSS):* We use the tree edit distance algorithm developed by Zhang and Shasha (Zhang, K. et al., 1989). This metric calculates the minimum cost to transform one ordered and labeled tree into another through a series of three fundamental operations:

  - **Insert:** Adds a new node (a class or a package) to the tree;
  - **Delete:** Removes a node from the tree;
  - **Rename (or Relabel):** Changes the label of a node. In our context, this is equivalent to moving a class from one package to another.

  The number of operations required to transform one tree into another represents the "distance" between these trees. A lower distance implies that the two hierarchical decompositions are similar. The comparison considers classes as leaves and packages as internal nodes.

  While MoJoFM is a standard metric for evaluating flat software clustering, it is less suited to comparing hierarchical structures as it primarily measures the number of move and join operations required to align two partitions, without considering the nested package structure. The ZSS tree edit distance, by contrast, is designed to compare tree structures, making it a more appropriate metric for evaluating the authoritativeness of the hierarchical decompositions.

- *Single-package commits ratio (SPCR):* This metric calculates the percentage of commits in which all modified classes belong to a single package, according to a given decomposition, to the number of commits. A higher ratio suggests that the algorithm grouped classes that evolve together. A "good" clustering should group classes that change together into the same package, thereby maximizing this value and reflecting better modularity from a maintenance perspective;

- *Average number of packages affected per commit (APC)*: Complementary to the ratio above, this metric measures the mean number of packages modified in each commit across the project's history. While the previous ratio indicates how often changes are perfectly localized to a single package, this average provides a broader view of how many packages these modifications are being distributed across.

The *average number of classes per package* and *maximum tree depth* metrics used in the experimental analysis reported in Chapter 3 are not used in this experiment because the first is linearly dependent on the *number of packages* metric and the second is indirectly captured by the *tree edit distance* metric.

4.5.2 Illustrative Example: Metrics Calculation

To provide a clear understanding of the proposed evaluation framework, this section applies the metrics to the hypothetical system introduced in Figure 4.1 along with an alternative decomposition hypothetically proposed by the optimization algorithm (Figure 4.4). Furthermore, Figure 4.5 shows the hierarchical decomposition trees for both solutions.

Number of Packages (NP): The scale of the decomposition is determined by counting every internal node (package) in the decomposition tree.

- HMD **decomposition:** Organizes classes into a root c_0, containing c_1 (parent of c_{1-1}) and c_2. Therefore, we have four packages, $\{c_0, c_1, c_2, c_{1-1}\}$, and $NP = 4$;
- DEV **decomposition:** Organizes classes into a root c_0, with child packages c_1 (parent of c_{1-1}) and c_2 (parent of c_{2-1}). Therefore, we have five packages, $\{c_0, c_1, c_2, c_{1-1}, c_{2-1}\}$, and $NP = 5$.

Tree Edit Distance (ZSS): The Zhang-Shasha algorithm quantifies the effort required to transform the algorithmic solution (HMD) into the authoritative design (DEV). In this

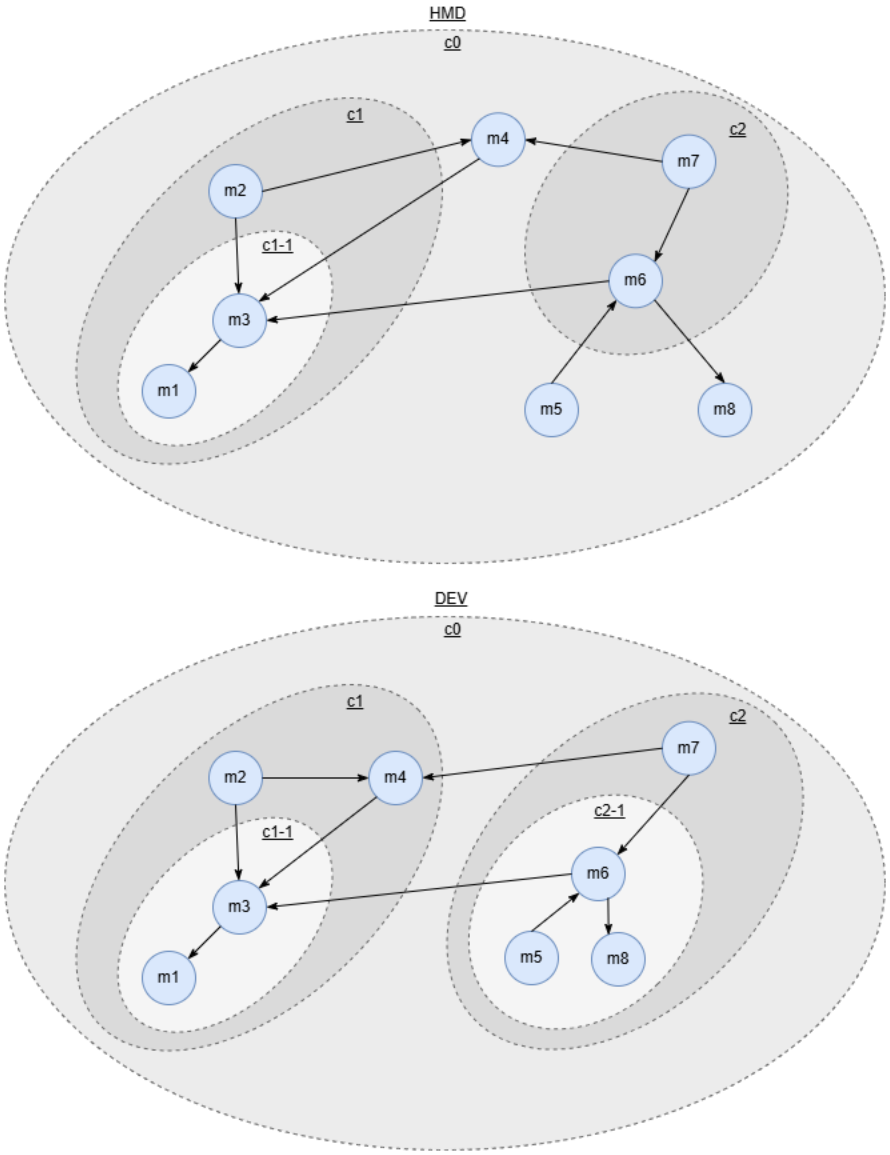


Figure 4.4: Comparison between the authoritative decomposition (*DEV*) and an algorithmic solution (*HMD*).

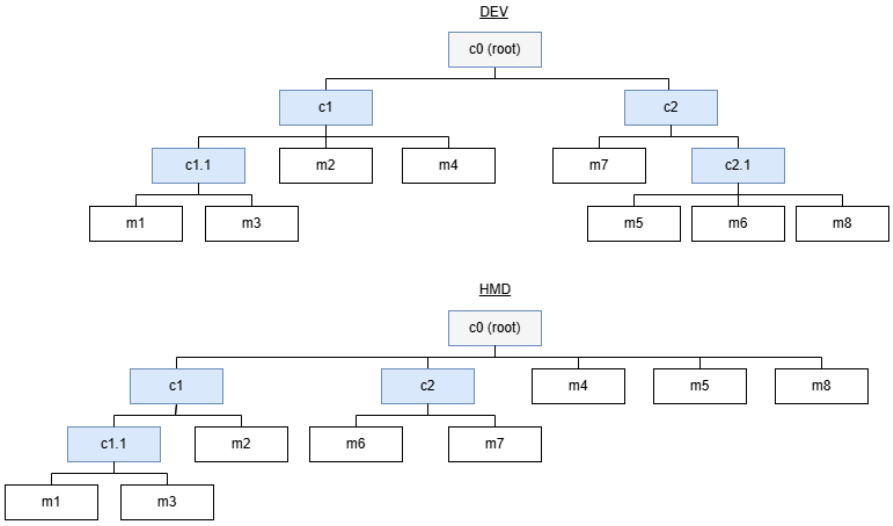


Figure 4.5: Comparison between the authoritative decomposition tree (*DEV*) and an algorithmic solution tree (*HMD*).

example, the transformation requires **7 operations**, resulting in a distance of **7**, as detailed in Table 4.5.

Table 4.5: ZSS Transformation Operations ($HMD \rightarrow DEV$)

Step	**Operation**	**Description**
1	DELETE	Remove m_4 from c_0
2	INSERT	Add m_4 as a child of c_1
3	DELETE	Remove m_5 from c_0
4	INSERT	Add m_5 as a child of c_{2-1}
5	DELETE	Remove m_8 from c_0
6	INSERT	Add m_8 as a child of c_{2-1}
7	INSERT	Add c_{2-1} as a child of c_2 (“adopts” m_6)

The Zhang-Shasha algorithm (ZSS) identifies the minimum-cost sequence of edit operations to transform the HMD tree into the DEV tree, strictly enforcing that ancestor-descendant relationships are preserved “if and only if” they existed in the source tree. In this specific case, the relocation of nodes m_4, m_5, and m_8 needs a break in their original hierarchy relative to the root c_0, as they move into new sub-trees. The relocation of node

m_6 from a direct child of c_2 to a child of the newly inserted c_{2-1} is a formal structural consequence of the insertion operation itself.

According to the definition of tree insertion (Zhang, K. et al., 1989, Section 2.1), adding a node n as the child of n' automatically makes n the parent of a "consecutive subsequence" of n''s existing children. Consequently, when c_{2-1} is inserted under c_2 in Step 7, the algorithm defines a subsequence including m_6, causing c_{2-1} to "adopt" it as a descendant. This allows m_6 to reach its target position as an atomic side effect of the insertion of its new parent. Since m_7 is already a child of c_2, it is not touched.

Single-package commits ratio (SPCR): Table 4.6 shows the class-to-package mapping for each decomposition, while Table 4.7 presents the detailed analysis of the 20 commits. A checkmark (✓) indicates that all classes in the commit belong to a single package in that specific decomposition.

Table 4.6: Class-to-Package mapping for the DEV and HMD decompositions.

Class	DEV Package	HMD Package
m1, m3	c1-1	c1-1
m2	c1	c1
m4	c1	**c0**
m5, m8	**c2-1**	**c0**
m6	**c2-1**	**c2**
m7	c2	c2

- *HMD* **decomposition:** With zero commits affecting classes from a single package out of 20 commits, the SPCR for this decomposition is **0.0%**;
- *DEV* **decomposition:** With three commits affecting classes from a single package out of 20 commits, the SPCR for this decomposition is **15.0%**.

Average number of packages affected per commit (APC): Table 4.7 shows the packages involved in each of the 20 commits for both the *DEV* and *HMD* decompositions. We count 44 packages in the *DEV* distribution and 49 packages in the *HMD* distribution. Therefore, we have:

Table 4.7: Commit-by-Commit analysis for the DEV and HMD decompositions.

#	Classes	DEV Packages Involved	DEV Single?	HMD Packages Involved	HMD Single?
01	All	{c1-1, c1, c2-1, c2}	–	{c1-1, c1, c0, c2}	–
02	m3, m4	{c1-1, c1}	–	{c1-1, c0}	–
03	m6, m8	{c2-1}	✓	{c2, c0}	–
04	m1, m5, m7	{c1-1, c2-1, c2}	–	{c1-1, c0, c2}	–
05	m1, m3, m4	{c1-1, c1}	–	{c1-1, c0}	–
06	m3, m5, m7	{c1-1, c2-1, c2}	–	{c1-1, c0, c2}	–
07	m1, m3, m7	{c1-1, c2}	–	{c1-1, c2}	–
08	m1, m3, m4	{c1-1, c1}	–	{c1-1, c0}	–
09	m2, m3, m4	{c1, c1-1}	–	{c1, c1-1, c0}	–
10	m5, m6	{c2-1}	✓	{c0, c2}	–
11	m2, m7	{c1, c2}	–	{c1, c2}	–
12	m1, m6, m8	{c1-1, c2-1}	–	{c1-1, c2, c0}	–
13	m5, m6, m8	{c2-1}	✓	{c0, c2}	–
14	m1, m2, m7	{c1-1, c1, c2}	–	{c1-1, c1, c2}	–
15	m1, m2	{c1-1, c1}	–	{c1-1, c1}	–
16	m2, m6, m7	{c1, c2-1, c2}	–	{c1, c2}	–
17	m5, m7, m8	{c2-1, c2}	–	{c0, c2}	–
18	m1, m2, m5	{c1-1, c1, c2-1}	–	{c1-1, c1, c0}	–
19	m2, m3, m4	{c1, c1-1}	–	{c1, c1-1, c0}	–
20	m6, m7, m8	{c2-1, c2}	–	{c2, c0}	–

- *HMD* **decomposition:** APC = $49/20 = 2.45$;

- *DEV* **decomposition:** APC = $44/20 = 2.20$.

The comparison between the *DEV* and *HMD* decompositions through SPCR and APC metrics highlights the impact of modular organization on maintenance effort. In this hypothetical scenario, the *DEV* structure proved superior for SPCR, achieving a measure of 15.0% and containing three distinct maintenance tasks (commits 03, 10, and 13) within a single package. Conversely, there were no commits (0.0%) involving a single package in the *HMD* decomposition, which would require developers to cross package boundaries for every task in the history.

This discrepancy is further reinforced by the APC metric, which indicates that *DEV* affects fewer packages on average (2.20) than *HMD* (2.45). In this example, the *DEV* design groups the classes that change together more effectively, involving fewer packages per commit.

4.5.3 Problem Instance Selection

We have selected a set of software systems for this study. They were selected based on the following criteria:

- *Open-source software*: we chose open-source libraries due to the availability of the source code, which is necessary to execute and evaluate the proposed approach;
- *Source code is available through a Git repository*: because we needed to evaluate project change history, we selected systems that store their source code in Git repositories. Git is one of the most popular version control systems in use today;
- *Source code is primarily written in Java language*: we decided to evaluate Java-based systems due to the availability of tools to extract project metadata automatically from the compiled code;
- *Different numbers of classes and packages*: to evaluate the proposed approach in different scenarios and scales, we picked systems with varying sizes, measured as the number of classes and packages (from tens to a few hundreds);
- *Usage and recent activity*: we also considered the number of stars and forks, as well as the last update date. We excluded projects with fewer than 1,000 stars and last updated more than 1 year ago. We also discarded forks, as they received little community attention.

Table 4.8 presents the selected systems, their descriptions, versions, and Git repository links. We chose to work with a limited number of classes and packages for the studies reported in this chapter because of the time required to run the Hierarchical Agglomerative Clustering (*HAC*) algorithm on large instances with many modules and edges, particularly when the *MDL* fitness function was used. Therefore, we extracted sub-modules with up to 200 classes from each of the selected systems. The limit was set to ensure that each instance's optimization could run for up to 5 hours. Table 4.9 shows the selected sub-modules and their number of classes and packages.

Table 4.8: Investigated software systems

Software system	Description	Version	Repository
AEP Core	A core library providing interfaces for managing and deploying applications to Google App Engine.	0.10.0	https://github.com/GoogleCloudPlatform /appengine-plugins
Elastic Search	A free, distributed, and RESTful search engine.	9.1.0	https://github.com/elastic/elasticsearch
JavaCC	A generator for building parsers from grammars.	7.0.14	https://github.com/javacc/javacc
JavaGeom	Java library to handle geometric calculations.	0.11.3	https://github.com/dlegland/javaGeom
Joda Money	Java library to represent monetary amounts.	2.0.3	https://github.com/JodaOrg/joda-money
JUnit	The 5th major version of the programmer-friendly testing framework for Java.	5.10.1	https://github.com/junit-team/junit5
JMetal	Java library for multi-objective optimization with metaheuristics.	6.2.2	https://github.com/jMetal/jMetal
JGit	An implementation of the Git VCS in Java.	6.8.0	https://github.com/eclipse-jgit/jgit
Log4J	A logging API and backend for Java.	2.21.1	https://github.com/apache/logging-log4j2
Lucene	Apache Lucene open-source search software.	11.0.0	https://github.com/apache/ lucene

The initial pool of candidate systems was identified through established research in Software Module Clustering (Pinto, 2014; Monçores et al., 2018). However, because our methodology relies on change history data to construct the *CRG*, we filtered the list to include only projects with accessible GitHub version control history. To ensure a sufficiently diverse and active dataset, we expanded our search to include GitHub-hosted projects that met the technical requirements described above, concerning programming language (Java), project size, and community activity. This refinement process led to the final selection of the 21 sub-modules detailed in Table 4.9.

Table 4.9: Investigated systems/sub-modules ordered by the number of classes.

ID	Software system	Sub-module	Classes	Packages
LUC	Lucene	Codecs	207	8
JCC	JavaCC	-	190	6
LJA	Log4J	API	181	7
LUH	Lucene	Highlighter	166	4
AEP	AEP	Core	162	17
JGE	JavaGeom	-	159	20
LJL	Log4J	Layout	150	3
JPE	JUnit	Platform Engine	135	9
JJE	JUnit	Jupiter Engine	129	8
JGP	JGit	PGM	121	5
JPL	JUnit	Platform Launcher	115	6
ELN	Elastic Search	Native	66	4
ELC	Elastic Search	Core	63	3
JGH	JGit	HTTP	61	3
ELG	Elastic Search	Geo	58	3
JML	JMetal	Lab	54	13
LUN	Lucene	Nori	47	3
JGL	JGit	LFS	41	4
JMO	Joda Money	-	28	2
LJI	Log4J	IO Streams	26	2
LJJ	Log4J	JPA	23	2

It is important to note the evolution of the dataset between the initial structural study presented in Chapter 3 and the change-history evaluation conducted in this chapter. Three sub-modules analyzed in Chapter 3 – specifically *JMA* (*JMetal Auto*), *JMC* (*JMetal Component*), and *JGS* (*JGit SSH*) – were not included in Table 4.9. While these instances provided valuable insights under static structural optimization (e.g., highlighting semantic domain boundaries in *JMA*), our hybrid methodology requires a sufficiently dense and active modification history to construct a meaningful Commit Relation Graph (CRG).

A preliminary analysis of the version-control logs revealed that *JMA*, *JMC*, and *JGS* exhibited relatively sparse co-modification matrices, dominated by single-class commits or structural re-organizations that would yield uninformative edge weights ($w(i,j) \approx 0$) across the percentile thresholds (CRG_p). So these sub-modules were

replaced with alternative components from the same parent projects – namely *JML* (*JMetal Lab*) and *JGL* (*JGit LFS*).

4.5.4 Comparative Analysis

To evaluate the proposed approach, we conducted a comparative analysis across five distinct experimental configurations. In this study, we distinguish between the *representation* of the software system and the *fitness function* that drives the HAC^* optimization. The scenarios are defined as follows:

- MQ **(Flat Baseline):** Represents the traditional non-hierarchical approach. It employs the *Modularization Quality* (*MQ*) fitness function to evaluate the system's Module Dependency Graph (*MDG*), as established in Chapter 3;

- HMD **(Hierarchical Structural Baseline):** Utilizes the complexity metric Ψ, based on the *Minimum Description Length* (*MDL*) principle, to evaluate a standard Hierarchical Module Decomposition representation. This configuration considers only the static structural dependencies of the source code;

- HMD^+_{25}**,** HMD^+_{50}**, and** HMD^+_{75} **(Enhanced HMD):** These configurations use the complexity metric Ψ (Equation 2.9) to evaluate an *Enhanced HMD* representation (HMD^+_p), which incorporates both static structural dependencies and modification dependencies derived from the Commit Relation Graph (*CRG*). The variants (HMD^+_{25}, HMD^+_{50}, and HMD^+_{75}) correspond to the percentile thresholds used to filter change-history associations.

The results from these configurations were compared with the *DEV* representation, the authoritative hierarchical decomposition proposed by the software developers. Next, we detail the results for each metric.

4.5.4.1 *Number of packages*

We present the raw data for the package counts across all investigated systems and configurations (*DEV*, *MQ*, *HMD*, and HMD_p^+) in Table 4.10. We observe large differences in the number of packages across solutions.

To statistically validate the observed differences, we performed a Kruskal-Wallis test across the six experimental configurations ($n = 126$ observations). The test yielded a significant result ($H(5) = 83.002,\ p < 0.001$), providing evidence that the choice of fitness function and system representation affects the scale of the resulting modular decomposition. This allows us to reject the null hypothesis that the distributions of package counts are identical across all algorithms, justifying a deeper analysis of the specific architectural biases inherent in each approach.

Furthermore, to illustrate the differences in the number of packages, we plotted a boxplot (Figure 4.6) and a heatmap (Figure 4.7). To assess performance relative to the *DEV* distributions in the heatmap, we evaluate each algorithm using a proportional *ratio*. The *Ratio*, displayed as numeric text in heatmap cells, represents the size of an algorithm's decomposition relative to the developer's baseline. A ratio of 1.0 indicates a perfect match in the number of packages. This is defined in Equation 4.6.

$$\text{Ratio} = \text{round}\left(\frac{\text{Packages}_{Algorithm}}{\text{Packages}_{DEV}}, 2\right) \tag{4.6}$$

The heatmap's color gradient visually emphasizes the magnitude of the differences in the number of packages – darker cells indicate a greater divergence from the ideal ratio of 1.0, regardless of whether the algorithm over-fragmented or under-approximated.

Table 4.10: Number of packages for the distributions of classes into packages produced by different algorithms for the selected instances.

Project	**DEV**	**MQ**	**HMD**	HMD^{+}_{25}	HMD^{+}_{50}	HMD^{+}_{75}
AEP	17	87	36	4	6	7
ELC	3	37	17	5	8	11
ELG	3	27	10	5	5	5
ELN	4	33	11	4	5	9
JCC	6	83	16	3	4	10
JGE	20	72	22	3	5	6
JGH	3	33	21	7	12	12
JGL	4	21	6	4	4	7
JGP	5	58	22	5	8	10
JJE	8	70	30	4	5	10
JML	13	26	11	5	4	6
JMO	2	13	5	4	5	5
JPE	9	69	28	5	5	9
JPL	6	60	26	6	8	10
LJA	7	95	39	5	5	9
LJI	2	15	11	4	4	4
LJJ	2	13	5	5	5	6
LJL	3	72	31	5	8	12
LUC	8	108	42	4	9	12
LUH	4	89	29	3	5	8
LUN	3	25	13	6	7	8
Min	2	13	5	3	4	4
Mean	6	53	21	5	6	8
Std. Dev.	5	30	11	1	2	2
Max	20	108	42	7	12	12

As illustrated in the boxplot (Figure 4.6), which shows the distribution of these ratios relative to the baseline, *MQ* generated significantly more packages than any other approach across all instances, with substantial variance and numerous high-end outliers. From an architectural perspective, the over-fragmentation in the *MQ* distributions – with an average of 53 packages compared to the developer's mean of 6 – is undesirable. This results in a "package-per-class" bias that undermines the system's navigability, as the algorithm prioritizes local cohesion so aggressively that it fails to recognize the broader subsystems intended by developers. This behavior was also observed in the previous study (Chapter 3). This fragmentation is visually reinforced in the heatmap (Figure 4.7), where the *MQ* column is dominated by dark cells indicating severe deviation from the baseline.

The purely structural hierarchical algorithm (*HMD*) reduced this over-fragmentation, yielding a mean of 21 packages. While an improvement over *MQ*, it still proposes nearly three times as many packages as found in the *DEV* distributions.

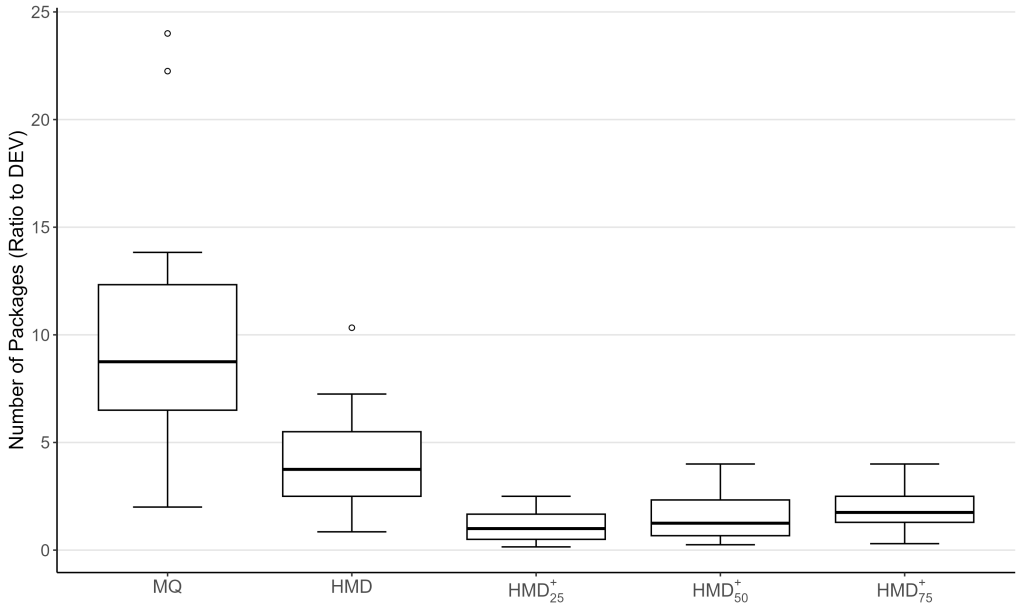


Figure 4.6: Boxplots illustrating the ratio of the number of packages generated by *MQ*, *HMD*, and HMD_p^+ relative to the *DEV* baseline.

This suggests that static code dependencies alone are often too granular to recover high-level architectural groupings.

The HMD_p^+ configurations produced fewer packages overall, as shown in the heatmap by predominantly light cells, with proportional ratios tightly clustered near 1.00. HMD_{25}^+ tended to slightly under-approximate the developer's design, with a mean of 5 packages. Notably, HMD_{50}^+ aligned perfectly with the average developer's design scale, yielding a mean of 6 packages. By selecting only the most frequent co-change relationships, these configurations successfully grouped classes into larger, more stable "core" packages without excessive fragmentation.

HMD_{75}^+ produced a mean of 8 packages, remaining close to the *DEV* structure while introducing a slight over-approximation. These results indicate that the inclusion of change-related information acts as a "consolidating force" in the optimization. By capturing logical dependencies that span multiple classes, the CRG_p allows the HMD_p^+ fitness functions to recognize that a simpler, more grouped hierarchy better describes the system's actual maintenance patterns. This leads to decompositions whose scale is closer

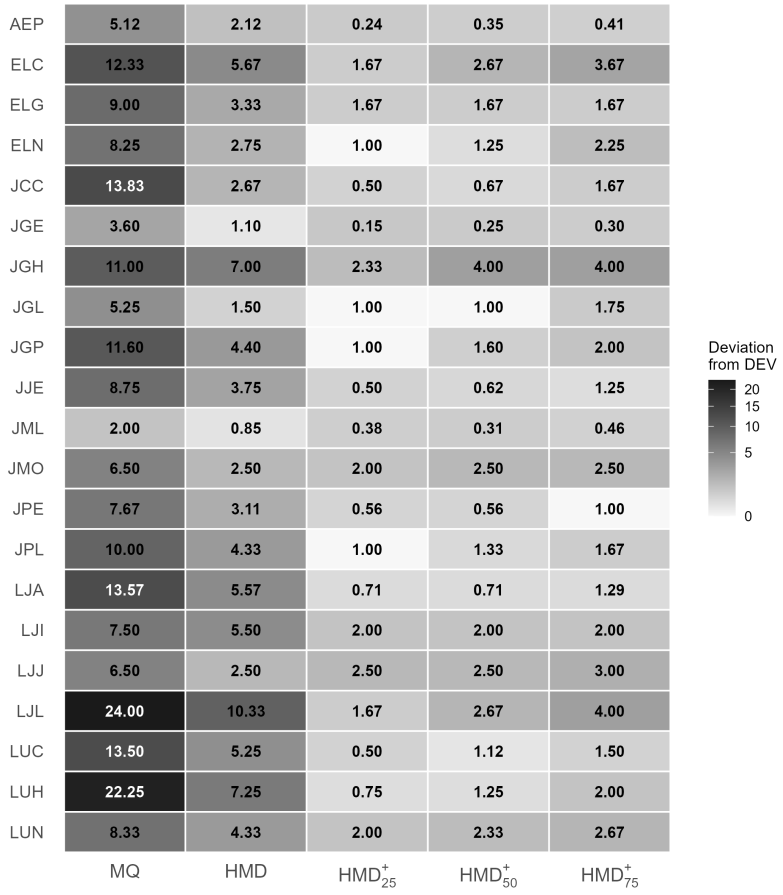


Figure 4.7: Heatmap illustrating the proportional ratio and deviation of generated packages relative to the *DEV* baseline.

to that of the developers' designs than the fragmented partitions produced by structural-only metrics.

A closer examination of the outliers in the HMD_p^+ configurations exposes some interesting characteristics. The boxplot and heatmap reveal several high-ratio outliers – most notably in projects such as JGH, ELC, and LJL, where HMD_{50}^+ produced ratios between 2.67 and 4.0. These high ratios might initially suggest that the developers created overly bloated "God Components" (e.g., LJL, which encompasses 150 classes in merely 3 packages). In the software architecture literature, a "God Component" (or "God Class") is a well-documented anti-pattern characterized by a module that centralizes excessive

system intelligence and responsibilities, violating the principle of cohesive design (Brown et al., 1998).

Attributing this entirely to developer error overlooks a gap in the algorithm's logic. Qualitative inspection of LJL reveals that the single package `org.apache.logging.log4j.layout.template.json.resolver` acts as a unified semantic API, clustering over 60 distinct resolution strategies. Although these resolvers evolve independently – prompting the change-driven fitness function to separate them – they are conceptually identical to the developer. In this instance, HMD_p^+ over-fragments a deliberately coarse-grained package, prioritizing co-change isolation over cohesion.

Conversely, the data show a mirror effect for the lower-bound outliers in projects such as JGE, AEP, and JML. For JGE (ratio of 0.25 under HMD_{50}^+), the system distributes 159 classes across 20 highly specific packages. While HMD_p^+ significantly compresses this architecture down to 5 packages to eliminate co-change dependencies, this aggregation exposes another limitation: vulnerability to cross-cutting concerns and global refactorings. Historical commits in JGE (such as commit $a2b861e7$, a large method-renaming task) show developers modifying up to 48 distinct packages simultaneously. The change history captures this transient coupling, tricking the fitness function into believing these distinct domains are architecturally close. By considering them, HMD_p^+ may distance itself from the original purpose of the code.

4.5.4.2 ZSS: Edit-distance Tree Algorithm

The results for the Tree Edit Distance (*ZSS*) provide the most direct measure of the authoritativeness of the proposed decompositions. By calculating the minimum number of operations (insertions, deletions, and relabeling) required to transform an automated solution into the developer's original design, we quantify the "descriptive quality" of each fitness function. A lower edit distance indicates the solutions are closer to each other. Table 4.11 presents these absolute distances, and Figure 4.8 illustrates the distribution across all 21 instances.

Table 4.11: Tree Edit Distance (ZSS) Comparison

Project	MQ	HMD	HMD^{+}_{25}	HMD^{+}_{50}	HMD^{+}_{75}
AEP	267	229	180	180	189
ELC	104	89	71	79	83
ELG	87	73	70	69	62
ELN	102	85	73	78	82
JCC	277	210	201	199	208
JGE	246	206	181	180	183
JGH	99	92	77	80	81
JGL	67	51	46	52	55
JGP	184	155	135	139	140
JJE	208	180	145	145	152
JML	94	83	78	75	74
JMO	44	38	34	34	36
JPE	214	173	150	155	158
JPL	181	155	129	132	133
LJA	283	238	195	195	203
LJI	45	44	38	34	36
LJJ	43	40	35	35	35
LJL	227	200	167	167	175
LUC	318	271	222	225	233
LUH	261	211	175	181	183
LUN	77	69	59	62	58
Min	43	38	34	34	35
Mean	163	138	117	119	122
Std. Dev.	92	75	62	62	65
Max	318	271	222	225	233

A Kruskal-Wallis test performed on the absolute *ZSS* distributions did not yield a statistically significant result ($H(4) = 6.08$, $p = 0.193$). However, this is expected due to the metric's unnormalized nature. Because *ZSS* measures absolute structural operations, its magnitude scales exponentially with the system size. Our dataset contains heterogeneous instances, ranging from small architectures (23 classes) to large sub-systems (207 classes). Consequently, the absolute variance between large and small projects hides the algorithmic variance in the non-parametric rank test, masking the practical improvements observed in individual instances.

Despite the statistical results, the data reveal a progression in approximating the *DEV* architectures. The non-hierarchical baseline (*MQ*) exhibited the lowest descriptive quality, with a mean distance of 163 operations. This high edit cost is a consequence of *MQ*'s flat partitioning bias; aligning an *MQ* solution with a nested *DEV* tree requires the algorithm to perform many insertions and deletions to reconstruct missing package

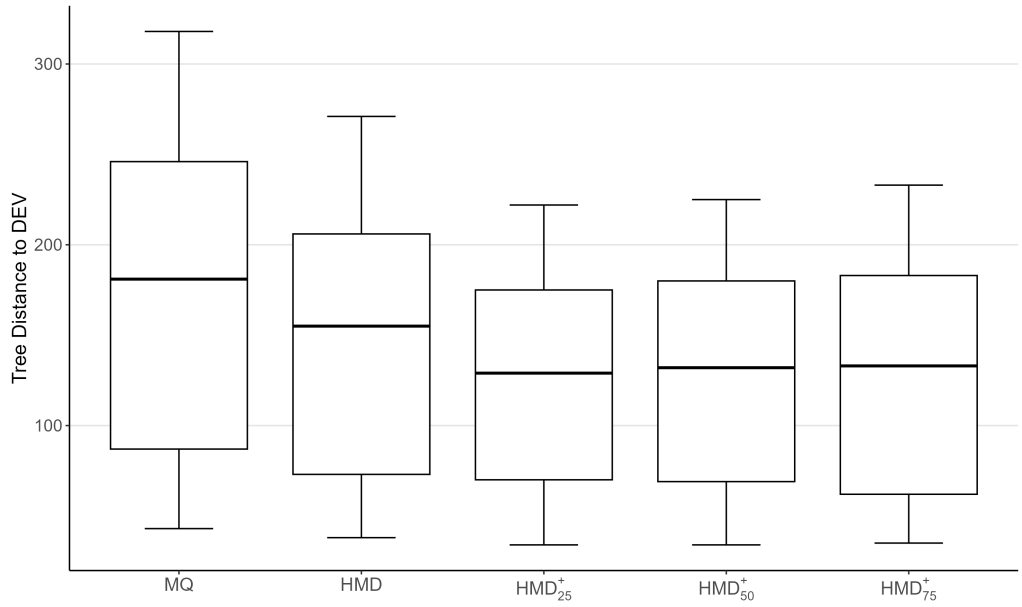


Figure 4.8: Tree Edit Distance (DEV x MQ x HMD x HMD_p^+) boxplot.

layers. The purely structural hierarchical baseline (HMD) mitigated this, reducing the mean distance to 138.

A key finding in this study is the improvement in authoritativeness achieved by incorporating change-related data. All configurations using HMD_p^+ outperformed the structural-only baselines. Notably, HMD_{25}^+ achieved the lowest mean distance (117), representing a 28% increase in descriptive quality compared to MQ and a 15% increase compared to HMD.

To accurately evaluate the algorithms while neutralizing the size variance that compromised the significance test, we analyzed the number of operations (or movements) per class in each project (ZSS/N), as shown in Table 4.12 and the distribution boxplot in Figure 4.9. Additionally, Figure 4.10 provides a heatmap of these ratios, enabling instance-by-instance comparison.

Table 4.12: Tree Edit Distance (ZSS) Comparison

Project	**MQ**	**HMD**	HMD^{+}_{25}	HMD^{+}_{50}	HMD^{+}_{75}
AEP	1.65	1.41	1.11	1.11	1.17
ELC	1.66	1.41	1.13	1.25	1.32
ELG	1.50	1.26	1.21	1.19	1.07
ELN	1.55	1.29	1.11	1.18	1.24
JCC	1.46	1.11	1.06	1.05	1.09
JGE	1.55	1.3	1.14	1.13	1.15
JGH	1.62	1.51	1.26	1.31	1.33
JGL	1.64	1.24	1.12	1.27	1.34
JGP	1.52	1.28	1.12	1.15	1.16
JJE	1.61	1.4	1.12	1.12	1.18
JML	1.75	1.54	1.44	1.39	1.37
JMO	1.56	1.36	1.21	1.21	1.29
JPE	1.58	1.28	1.11	1.15	1.17
JPL	1.57	1.35	1.12	1.15	1.16
LJA	1.56	1.31	1.08	1.08	1.12
LJI	1.72	1.69	1.46	1.31	1.38
LJJ	1.87	1.74	1.52	1.52	1.52
LJL	1.51	1.33	1.11	1.11	1.17
LUC	1.53	1.31	1.07	1.09	1.13
LUH	1.57	1.27	1.05	1.09	1.1
LUN	1.63	1.47	1.26	1.32	1.23
Min	1.46	1.11	1.05	1.05	1.07
Mean	1.60	1.37	1.18	1.20	1.22
Std. Dev.	0.09	0.15	0.14	0.12	0.12
Max	1.87	1.74	1.52	1.52	1.52

The MQ baseline has a mean of 1.60, indicating that reaching the developer's architecture requires more operations than there are classes in the system (mainly due to the flat solution). The HMD baseline improves this to 1.37. However, the HMD^{+}_{p} configurations lower this threshold, with HMD^{+}_{25} achieving a mean of 1.18. To formally validate the differences suggested by Table 4.12, we conducted a Friedman test across the five configurations ($n = 21$ systems, since each system was evaluated under every algorithm). The test yielded a highly significant result ($\chi^2(4) = 61.407,\ p < 0.001$), confirming that the normalized structural distance to the developer's architecture differs across algorithms. To identify which specific pairs differ, we performed pairwise Wilcoxon signed-rank tests with Holm correction for multiple comparisons, reporting the matched-pairs rank-biserial correlation (r) as the effect size for each pair. Table 4.13 summarizes these results.

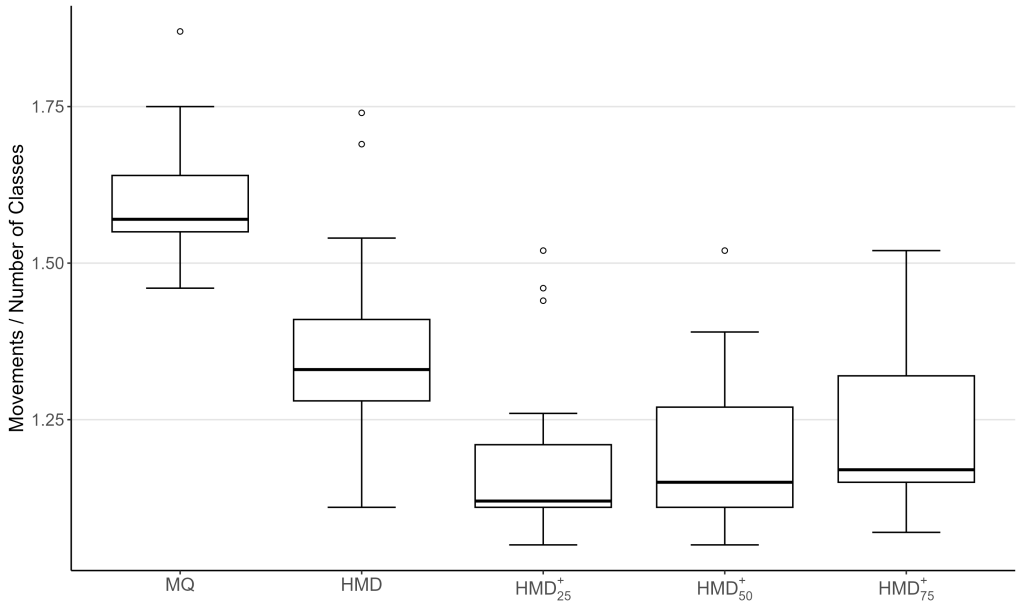


Figure 4.9: Tree Edit Distance ratio (Movements / Number of Classes) distribution across configurations.

Table 4.13: Pairwise comparisons of the normalized Tree Edit Distance (ZSS/N) across configurations (Wilcoxon signed-rank test, Holm-corrected, $n = 21$)

Comparison	W	p	**Median** Δ	r	p_{adj}	**Sig.**
MQ vs. HMD	230	< 0.001	0.038	0.991	< 0.001	Yes
MQ vs. HMD^{+}_{25}	231	< 0.001	0.076	1.000	< 0.001	Yes
MQ vs. HMD^{+}_{50}	231	< 0.001	0.070	1.000	< 0.001	Yes
MQ vs. HMD^{+}_{75}	231	< 0.001	0.069	1.000	< 0.001	Yes
HMD vs. HMD^{+}_{25}	228	< 0.001	0.045	0.974	< 0.001	Yes
HMD vs. HMD^{+}_{50}	226	< 0.001	0.038	0.957	< 0.001	Yes
HMD vs. HMD^{+}_{75}	224	< 0.001	0.036	0.939	< 0.001	Yes
HMD^{+}_{25} vs. HMD^{+}_{50}	78	0.313	0.000	-0.257	0.313	No
HMD^{+}_{25} vs. HMD^{+}_{75}	47	0.030	-0.010	-0.552	0.091	No
HMD^{+}_{50} vs. HMD^{+}_{75}	70	0.114	-0.005	-0.394	0.228	No

As shown in Table 4.13, every comparison involving MQ, as well as every comparison between HMD and an HMD^{+}_{p} configuration, remains statistically significant after correction, with effect sizes approaching the theoretical maximum ($r \geq 0.94$). This confirms that incorporating change-history data via the *CRG* yields a more structurally

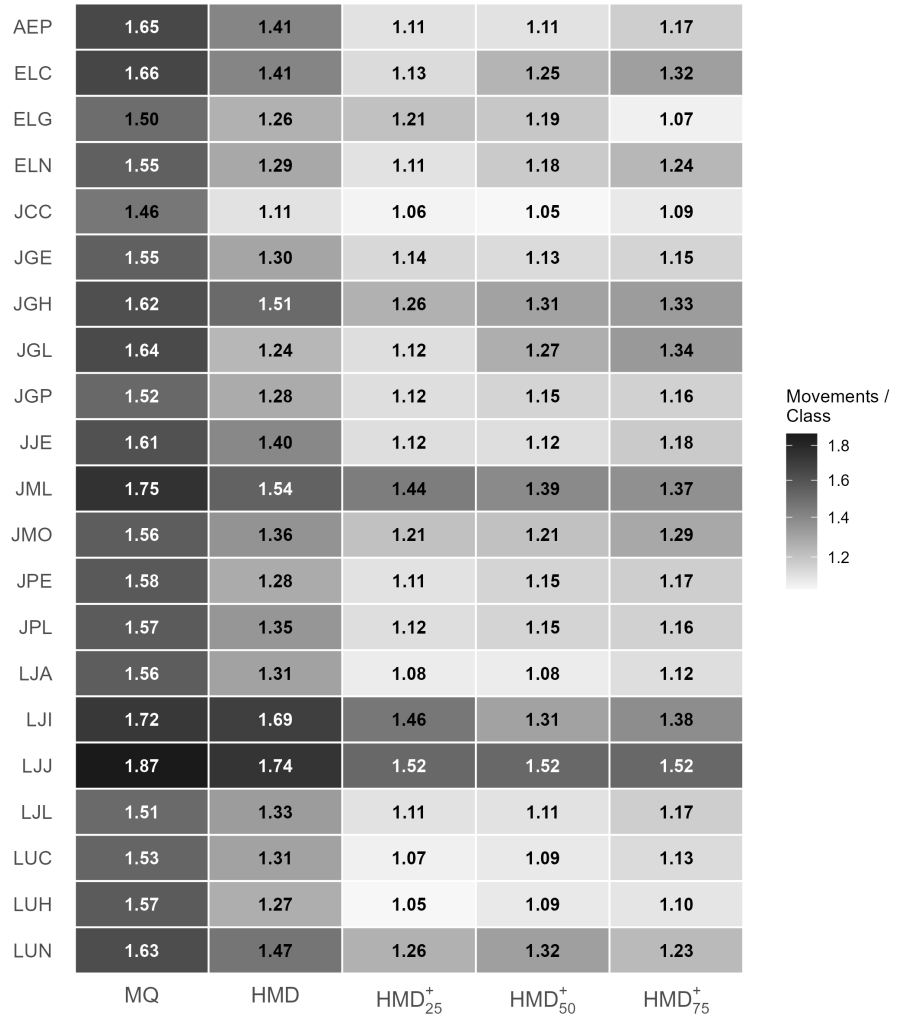


Figure 4.10: Tree Edit Distance ratio (Movements / Number of Classes) heatmap, illustrating relative algorithmic performance per instance.

authoritative decomposition than either the non-hierarchical baseline or the purely structural hierarchical model. In contrast, the pairwise comparisons among the three HMD_p^+ configurations themselves did not reach significance after correction, indicating that, on this specific metric, the choice of threshold does not produce a statistically distinguishable effect on structural fidelity to the developer's design.

The instance-level performance is visually reinforced by the heatmap (Figure 4.10). The MQ and HMD columns are dominated by darker cells, reflecting the high edit

cost of reconstructing nested developer architectures from purely structural or flat baselines. In contrast, the HMD_p^+ configurations exhibit consistently lighter cells across the vast majority of the dataset. This uniform transition to lighter shades confirms that the gain in authoritativeness is not limited to a few specific projects but is a systemic improvement across heterogeneous instances.

A closer examination of the outliers in the ZSS/N ratio distributions reveals critical boundaries between where the HMD_p^+ approach excels and where it still needs improvement. The data exposes a distinct vulnerability in micro-architectures. The highest edit distance ratios under HMD_{50}^+ are observed for the smallest systems in the dataset, notably LJJ (1.52) and LJI (1.31). For example, LJJ (Log4j JPA) consists of only 23 classes organized into two packages, while LJI (Log4j IO Streams) contains 26 classes across two packages. A qualitative inspection of their commit histories confirms that these sub-modules are highly specialized, atomic libraries.

LJJ, for instance, serves strictly as a bridge for writing Log4j events to a database using the Java Persistence API. Its classes are either core entities (*e.g.*, `JpaAppender`) or data translators (*e.g.*, `ContextMapAttributeConverter`). While semantically unified as a single domain for the developer, these different class types often exhibit distinct cycles of update – adding a new converter does not require modifying the core appender. The *MDL* fitness function of HMD_p^+ detects this independence and over-fragments these projects, yielding five distinct packages for LJJ. To transform the solution back into the developer's simple binary design, the *ZSS* algorithm must execute a high number of operations. This confirms that the change-driven hierarchical approach suffers somewhat with genuinely small systems, where static semantic boundaries are already compact enough to manage maintenance effectively without further subdivision.

Conversely, analyzing the absolute *ZSS* outliers for medium to large-scale systems – such as JML (1.39 ratio with 75 absolute operations) and LJL (1.11 ratio with 167 absolute operations under HMD_{50}^+) – reveals a different phenomenon. In the case of JML (jMetal Lab), the developers originally fragmented 54 classes across 13 highly specific packages based on technical roles (*e.g.*, studies, component, and util). However,

an analysis of the commit log reveals an odd behavior resulting from this technical partitioning, closely resembling "Shotgun Surgery" coupling. This phenomenon is classically defined as a code smell where a single conceptual change requires cascading, simultaneous modifications across multiple dispersed modules (Fowler, 1999).

For instance, large commits like $cf8176b8$ touched 27 different packages simultaneously because implementing or updating a single experimental feature required coordinated changes across `GenerateFriedmanTestTables.java`, `NSGAIIStudy.java`, and `ExperimentBuilder.java`. The change history showed that the original 13-package structure might have been too horizontally fragmented for the system's reality, prompting HMD_{50}^{+} to compress the architecture into four feature-aligned macro-components.

Because the *ZSS* metric strictly measures authoritativeness (proximity to the *DEV* baseline), it penalizes the algorithm for this difference. To align the optimized HMD_{50}^{+} tree with the *DEV* tree for JML, the edit-distance algorithm needs to re-fragment the classes into the 13 packages, which are tightly coupled from a feature perspective. A similar penalty occurs in LJL (Log4j Layout Template), where a component of 150 classes was divided into just three packages. A single package resolver alone contained 64 distinct files. While this package serves as a unified semantic API for the developer (as discussed previously), from a change-history standpoint, it acts as a maintenance monolith. Because capabilities like `MessageResolver` and `StackTraceElementResolver` have distinct life cycles, HMD_{50}^{+} decomposed this package into eight subsystems, which are cohesive with respect to actual maintenance life cycles – but penalized by the *ZSS* metric.

Therefore, a significant portion of the remaining edit distance in the HMD_{p}^{+} configurations does not necessarily indicate algorithmic failure but rather reflects a semantic gap. In projects where developers employ highly fragmented designs separated by technical layers (JML) or exceptionally coarse-grained packages (LJL), achieving a *ZSS* of zero is unlikely for a co-change-driven algorithm and not necessarily desirable. The remaining edit operations precisely quantify the tension between how developers

semantically categorize their code and how they actually modify it during software evolution.

4.5.4.3 Single-package commits ratio

The single-package commits ratio (SPCR) quantifies the localization of changes within the proposed package decompositions. A higher ratio indicates a decomposition in which the changes recorded in the analyzed history are more frequently contained within a single package. The results for this metric across all configurations are presented in Table 4.14.

To statistically validate the observed differences, we performed a Kruskal-Wallis test across the six experimental configurations. The test yielded a highly significant result ($H(5) = 24.056$, $p < 0.001$), providing evidence that the choice of clustering algorithm fundamentally alters the change history cohesion of the resulting architecture.

The structural baselines, *MQ* and *HMD*, consistently underperformed the developer's design. While the *DEV* structure achieved a mean ratio of 0.77, *MQ* and *HMD* achieved only 0.60 and 0.65, respectively. This is a direct consequence of the over-fragmentation observed in the package count analysis; by splitting the system into dozens of artificial modules based purely on static coupling, the algorithms artificially forced historical commits to span multiple boundaries.

The inclusion of change-history data via the HMD_p^+ configurations bridged this gap. Notably, HMD_{50}^+ achieved a mean ratio of 0.76, almost recovering the localization capacity of the original developer structure. Crucially, HMD_{50}^+ achieves this high SPCR while maintaining an average of six packages (matching the *DEV* baseline scale). This demonstrates that the algorithm is not aggregating the entire system into a single monolithic package, but is instead successfully identifying the boundaries that encapsulate real change-related tasks.

Table 4.14: Single-package commits ratio

Project	DEV	MQ	HMD	HMD^{+}_{25}	HMD^{+}_{50}	HMD^{+}_{75}
AEP	0.64	0.53	0.59	0.75	0.77	0.74
ELC	0.98	0.84	0.87	0.90	0.88	0.88
ELG	0.73	0.66	0.68	0.95	0.82	0.84
ELN	0.50	0.47	0.52	0.70	0.50	0.50
JCC	0.84	0.60	0.75	0.68	0.89	0.73
JGE	0.39	0.23	0.36	0.31	0.48	0.34
JGH	0.83	0.58	0.59	0.76	0.69	0.68
JGL	0.79	0.71	0.78	0.77	0.82	0.73
JGP	0.87	0.80	0.84	0.87	0.84	0.82
JJE	0.79	0.63	0.65	0.91	0.88	0.72
JML	0.69	0.49	0.63	0.82	0.79	0.76
JMO	0.87	0.58	0.71	0.72	0.75	0.76
JPE	0.80	0.64	0.77	0.72	0.88	0.79
JPL	0.75	0.60	0.64	0.74	0.86	0.70
LJA	0.85	0.75	0.80	0.83	0.88	0.85
LJI	0.94	0.47	0.47	0.55	0.61	0.51
LJJ	0.75	0.58	0.60	0.75	0.64	0.71
LJL	0.65	0.48	0.50	0.59	0.63	0.59
LUC	0.71	0.52	0.54	0.60	0.62	0.65
LUH	0.87	0.66	0.71	0.73	0.92	0.86
LUN	0.91	0.75	0.75	0.80	0.81	0.77
Min	0.39	0.23	0.36	0.31	0.48	0.34
Mean	0.77	0.60	0.65	0.74	0.76	0.71
Std. Dev.	0.14	0.14	0.13	0.14	0.13	0.13
Max	0.98	0.84	0.87	0.95	0.92	0.88

Furthermore, to illustrate these differences across the dataset, we plotted a box-plot (Figure 4.11) and a heatmap (Figure 4.12). To accurately assess performance relative to the *DEV* distributions in these visualizations, we evaluate each algorithm using a proportional ratio, similar to the analysis performed for the number of packages. The ratio represents the single-package commit ratio (*SPCR*) for a given algorithm, expressed as a proportion of the developer's baseline and rounded to two decimal places, as defined in Equation 4.7.

$$\text{Ratio} = \text{round}\left(\frac{\text{SPCR}_{Algorithm}}{\text{SPCR}_{DEV}}, 2\right) \tag{4.7}$$

A ratio of 1.0 indicates that the algorithm perfectly matched the commit localization capacity of the original developer structure. Values below 1.0 indicate that commits span more boundaries than originally intended, while values above 1.0 indicate that the algorithm successfully encapsulated more historical commits within single

packages than the original design. The heatmap's color gradient visually emphasizes this divergence from the ideal ratio of 1.0.

As illustrated in the box-plot (Figure 4.11), which shows the distribution of these ratios relative to the baseline, the structural approaches (*MQ* and *HMD*) exhibit a localization deficit. Conversely, the HMD_p^+ distributions shift upward to strongly align with the 1.0 median.

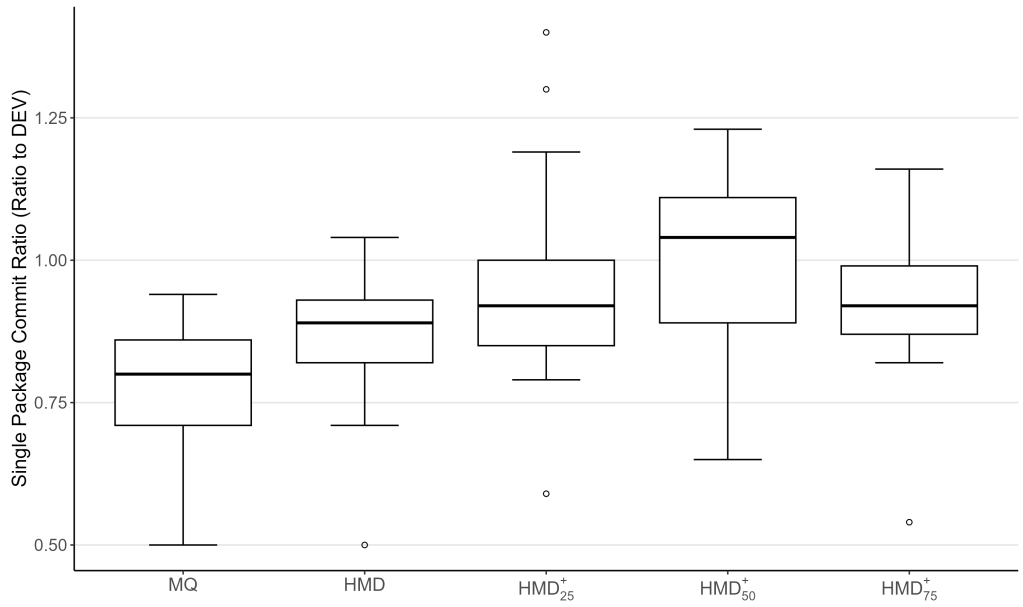


Figure 4.11: Single-package commits ratio boxplot, illustrating the proportional ratio generated by *MQ*, *HMD*, and HMD_p^+ relative to the *DEV* baseline.

An examination of the dataset's outliers reveals how some system-level architectural decisions influence commit localization. The lowest absolute SPCR values are observed in JGE, where the *DEV* baseline achieves only 0.39, and HMD_{50}^+ reaches 0.48. As established in previous sections, JGE is a hyper-fragmented system in which even a single commit – such as a global refactoring – can affect up to 48 packages simultaneously. Because the project's commit history is inherently dispersed across the entire codebase, no partitioning algorithm – human or automated – can achieve a high single-package ratio without completely flattening the system.

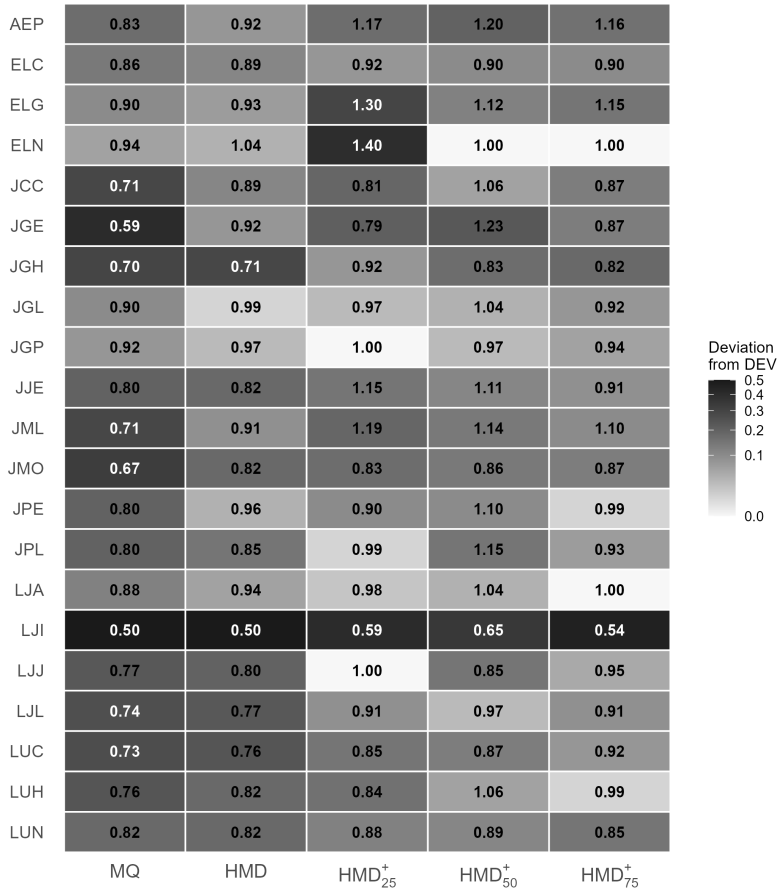


Figure 4.12: Single-package commits ratio heatmap, illustrating the proportional ratio of algorithmic localization relative to the *DEV* baseline.

Conversely, the highest SPCR values are observed for ELC (0.98 under *DEV*) and LJI (0.94 under *DEV*). Both instances are highly cohesive, atomic micro-libraries containing 3 or fewer packages. In these tightly scoped environments, nearly every commit is localized by default, establishing an artificially high ceiling that the algorithms effortlessly approach (*e.g.*, HMD^{+}_{25} scores 0.90 for ELC). Furthermore, the standard deviation of the SPCR metric across the 21 instances remains remarkably stable for all configurations (ranging only from 0.13 to 0.14). This stability suggests that integrating the *CRG* improves the overall mean localization without introducing volatility or extremes across heterogeneous projects.

Finally, an analysis of the heatmap reveals instances where the HMD_p^+ configurations actually *exceed* the *DEV* baseline in commit localization (Ratio > 1.0). For example, in AEP, the *DEV* SPCR of 0.64 was improved to 0.77 by HMD_{50}^+; similarly, JML improved from 0.69 to 0.79. In these cases, the proposed decompositions would have localized more of the past commits than the developers' design. Since the *CRG* is built from these same commits, this result is partly expected, as the optimization explicitly favors grouping classes that were modified together. Still, it shows that the HMD_p^+ approach does not merely reproduce the developers' design, and the differences between both may indicate package boundaries worth reviewing.

4.5.4.4 Average number of packages per commit

The average number of packages affected per commit (APC) serves as a complementary metric to the single-package commits ratio. While the SPCR measures perfect encapsulation (commits affecting exactly one package), the APC quantifies the overall dispersion of maintenance effort across the architecture. A lower average indicates that, even when commits cross package boundaries, the changes are spread over fewer packages.

The raw data and statistical summary for this metric are presented in Table 4.15. To statistically validate the observed differences, we performed a Kruskal-Wallis test across the six experimental configurations. The test yielded a highly significant result ($H(5) = 59.861$, $p < 0.001$), providing strong evidence that the architectural boundaries defined by the different algorithms dictate the dispersion of changes.

The structural baselines demonstrate a marked deficiency in change localization. *MQ* produced an average of 3.00 packages per commit, more than twice the developer's mean of 1.43. This confirming evidence suggests that the fragmentation inherent in non-hierarchical structural clustering forces changes to span multiple packages, thereby increasing architectural complexity. While the hierarchical *HMD* baseline showed improved localization (Mean = 2.16), it still resulted in a 51% higher dispersion than the developer-authored decompositions.

Table 4.15: Average number of packages per commit

Project	DEV	MQ	HMD	HMD^{+}_{25}	HMD^{+}_{50}	HMD^{+}_{75}
AEP	1.71	3.04	2.27	1.28	1.27	1.35
ELC	1.02	1.48	1.27	1.13	1.23	1.25
ELG	1.30	3.04	1.98	1.05	1.23	1.20
ELN	1.93	4.85	2.80	1.73	1.93	2.38
JCC	1.23	3.40	1.61	1.38	1.14	1.55
JGE	3.26	7.45	3.28	1.94	1.89	2.03
JGH	1.23	2.82	2.57	1.43	1.81	1.83
JGL	1.29	1.76	1.38	1.37	1.24	1.43
JGP	1.17	1.75	1.45	1.17	1.24	1.28
JJE	1.37	2.54	2.12	1.10	1.16	1.53
JML	1.67	2.82	1.65	1.22	1.25	1.35
JMO	1.13	2.14	1.58	1.45	1.44	1.41
JPE	1.35	2.77	1.92	1.32	1.16	1.37
JPL	1.36	2.66	2.10	1.34	1.22	1.48
LJA	1.26	1.80	1.55	1.20	1.15	1.20
LJI	1.06	4.23	4.18	1.71	1.59	1.80
LJJ	1.25	2.09	1.63	1.50	1.58	1.55
LJL	1.50	5.17	3.96	1.63	1.78	2.11
LUC	1.54	3.00	2.66	1.45	1.51	1.61
LUH	1.16	2.36	1.80	1.30	1.11	1.20
LUN	1.14	1.91	1.68	1.40	1.37	1.49
Min	1.02	1.48	1.27	1.05	1.11	1.20
Mean	1.43	3.00	2.16	1.39	1.40	1.54
Std. Dev.	0.48	1.42	0.82	0.22	0.27	0.32
Max	3.26	7.45	4.18	1.94	1.93	2.38

The change-driven configurations HMD^{+}_{25} and HMD^{+}_{50} achieved average counts (1.39 and 1.40, respectively) slightly lower than that of the developers' designs (1.43), and HMD^{+}_{50} did so while keeping an average scale of six packages, similar to the DEV baseline (Section 4.5.4.1). Because the optimization explicitly favors grouping classes modified together in the same history on which APC is computed, this result shows that the optimization achieved its intended effect, rather than that the generated designs are superior to those authored by the developers.

Furthermore, the HMD^{+}_{p} configurations exhibited greater stability across the systems, with a standard deviation for HMD^{+}_{25} (0.22) less than half that of the *DEV* baseline (0.48). This indicates that APC varied less across systems for the HMD^{+}_{p} configurations than for the developers' designs. For instance, in the hyper-fragmented *JGE* instance, the developer's architecture yielded an APC of 3.26, and *MQ* worsened this to 7.45. HMD^{+}_{50} reduced it to 1.89.

To visually assess this performance relative to the *DEV* baseline across the dataset, we plotted a box-plot (Figure 4.13) and a heatmap (Figure 4.14). To neutralize the inherent variance in absolute dispersion between highly cohesive and highly fragmented projects, both visualizations utilize a proportional ratio to the *DEV* distributions, defined in Equation 4.8:

$$\text{Ratio} = \text{round}\left(\frac{\text{APC}_{Algorithm}}{\text{APC}_{DEV}}, 2\right) \tag{4.8}$$

Crucially, because a lower Average Number of Packages per Commit (APC) indicates better change localization, a ratio below 1.0 is desirable. Values below 1.0 indicate that the algorithm successfully reduced the dispersion of commits compared to the developer's design, whereas values above 1.0 indicate increased spread.

As illustrated in the boxplot (Figure 4.13), which plots the distribution of these ratios, the structural approaches (*MQ* and *HMD*) suffer from dispersion, with medians well above the 1.0 baseline and numerous high-end outliers. Conversely, the HMD_p^+ configurations effectively compress this spread, pulling the distributions downward and frequently causing them to fall below the *DEV* threshold.

The heatmap (Figure 4.14) visually reinforces this consistency on a per-instance basis. The HMD_{25}^+ and HMD_{50}^+ configurations produced ratios below 1.0 in 10 and 12 of the 21 instances, respectively. This indicates that integrating the Commit Relation Graph leads the algorithm to propose boundaries that reduce the dispersion of the analyzed commits across packages.

An examination of the outliers in the APC distribution highlights the complex relationship between structural decomposition and maintenance dispersion. The most extreme upper-bound outlier is *JGE*, a system previously identified as an outlier in previous metrics. In the developer's original design, the average commit touched 3.26 packages – a highly dispersed maintenance reality. The non-hierarchical *MQ* algorithm aggravated this architectural degradation, shattering the system and driving the dispersion to 7.45 packages per commit. However, the HMD_{50}^+ configuration reduced this fragmentation, compressing the dispersion down to 1.89. In this instance, the

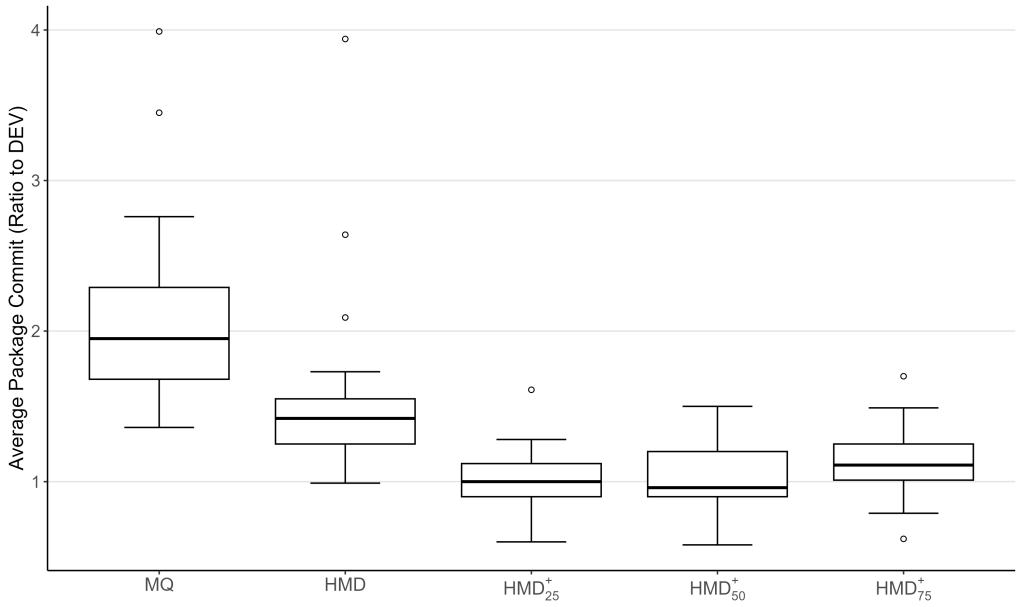


Figure 4.13: Average number of packages per commit boxplot, illustrating the proportional dispersion ratio generated by *MQ*, *HMD*, and HMD_p^+ relative to the *DEV* baseline.

change-aware optimization produced a decomposition in which the analyzed commits involve classes residing in fewer packages than in the developers' design.

Conversely, analyzing outliers in which HMD_p^+ mathematically "underperformed" the developer's baseline reveals the deceptive nature of the APC metric when applied to very large components. For instance, in *LJL* (Log4j Layout), the developer's baseline achieved a relatively low APC of 1.50, while HMD_{50}^+ yielded 1.78. While this appears to be a localized regression, cross-referencing it with the package count data reveals the developer achieved a low APC by dumping 150 classes into just three packages. This low dispersion is partly a consequence of concentrating many classes in few packages. HMD_{50}^+ decomposed LJL into eight packages, and the increase in APC to 1.78 reflects this finer decomposition: commits previously contained within a single large package now cross the boundaries of the new packages. Whether this finer decomposition is preferable to the developers' one would require their assessment.

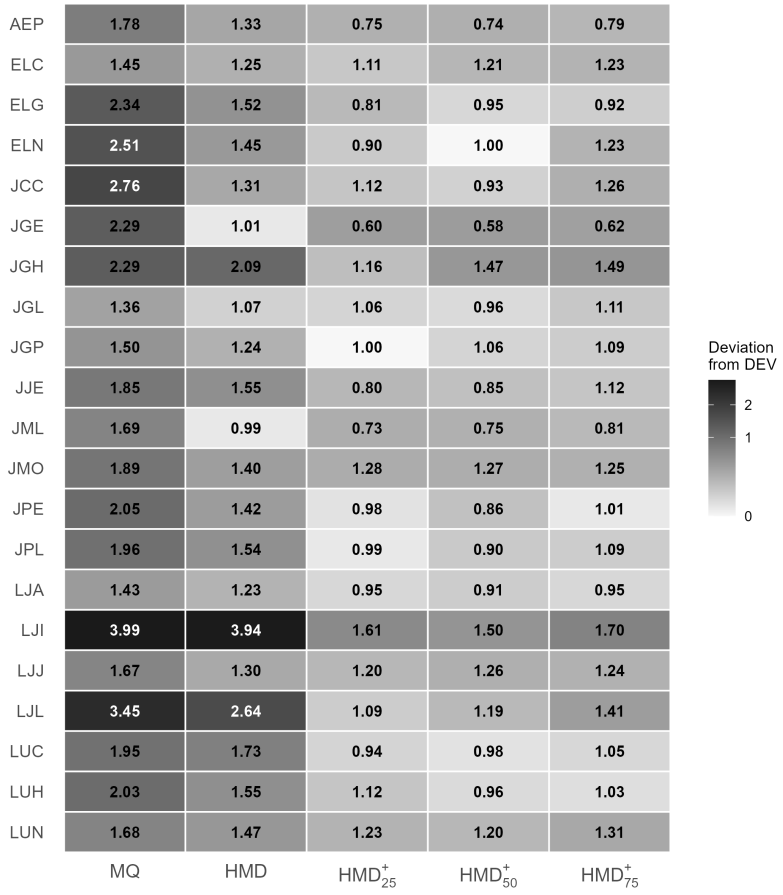


Figure 4.14: Average number of packages per commit heatmap, indicating algorithmic dispersion relative to the *DEV* baseline (Ratio < 1.0 indicates improvement).

Finally, the data exposes the algorithm's inherent lower-bound limitation when applied to atomic micro-architectures. Projects like LJI (Log4j IO Streams) and ELC (Elasticsearch Core) boast near-perfect developer APC scores of 1.06 and 1.02, respectively. Because these systems are highly specialized, atomic libraries (containing only 2 or 3 packages), virtually every commit is perfectly encapsulated by default. However, because the MDL-based fitness function is mathematically driven to seek optimized hierarchical solutions, it over-fragments these domains. By splitting LJI into 4 or 5 packages, the algorithm forces the commits to cross newly created boundaries, raising the APC to 1.59 under HMD^{+}_{50}. This reiterates that while highly effective at

curing large-scale dispersion, forcing change-driven hierarchical clustering onto legitimately atomic micro-libraries might introduce unnecessary inner packages.

4.5.5 Discussion

The quantitative evaluation across the 21 software instances shows that while purely structural metrics often struggle with over-fragmentation and maintenance dispersion, integrating the Commit Relation Graph (*CRG*) into the optimization process significantly improves both the authoritativeness (*ZSS*) and the change localization (SPCR and APC) of the resulting architectures.

The evaluated configurations allow the overall gain over the flat baseline to be decomposed into two consecutive steps. The first step, from *MQ* to *HMD*, changes both the representation (from flat to hierarchical) and the fitness function (from *MQ* to Ψ), and reduces the mean normalized edit distance (ZSS/N) from 1.60 to 1.37. The second step, which adds the CRG-derived edges, further reduced it to 1.18, 1.20, and 1.22 for HMD^{+}_{25}, HMD^{+}_{50}, and HMD^{+}_{75}, respectively (Table 4.12). Along this sequence, between 54% and 60% of the overall reduction is associated with the hierarchical MDL-based formulation, and between 40% and 46% with the change-history data.

Because HMD and the HMD^{+}_{p} configurations share the same representation, fitness function, and search algorithm and differ only in the CRG-derived edges, the significant differences between them (Table 4.13) can be attributed to the change-history data. This attribution, however, is conditional on the hierarchical formulation: the *CRG* was not evaluated in a flat formulation (*e.g.*, by adding its edges to the *MDG* optimized by *MQ*), and the first step does not separate the effect of the hierarchical representation from that of the Ψ metric. Therefore, the two contributions should not be interpreted as independent, and our results do not indicate whether the change-history data would yield similar gains without the hierarchical representation.

To examine whether the gains depend on the characteristics of the systems, we computed the Pearson correlation coefficient (ρ) between the improvement provided by each configuration and three characteristics of the systems: the number of classes, the

single-package commits ratio (SPCR), and the average number of packages per commit (APC) of the developers' designs. Since absolute edit distances grow with system size (Section 4.5.4.2), the improvement is measured on the normalized distance relative to a baseline, $\Delta ZSS/N = (ZSS_{Alg} - ZSS_{Base})/N$. A negative $\Delta ZSS/N$ indicates an improvement in approximating the DEV solution; therefore, a negative correlation means the improvement tends to be larger for higher values of the characteristic. Table 4.16 summarizes the results.

Table 4.16: Correlation (Pearson, n = 21) between the normalized structural improvement ($\Delta ZSS/N$) and system characteristics.

Configuration	**Classes**	**SPCR (DEV)**	**APC (DEV)**
HMD^{+}_{25} vs. MQ	**-0.527**	0.011	-0.035
HMD^{+}_{50} vs. MQ	**-0.710**	0.063	-0.122
HMD^{+}_{75} vs. MQ	**-0.609**	0.079	-0.114
HMD vs. MQ	**-0.417**	0.140	-0.130
HMD^{+}_{25} vs. HMD	**-0.137**	-0.142	0.104

They give the following insights regarding the approximation with the *DEV* architectures:

System size. Relative to the flat baseline, the improvement of the HMD^{+}_{p} configurations shows a negative correlation with the number of classes (from -0.53 to -0.71, significant at the 5% level), indicating that, within the range of evaluated sizes (23 to 207 classes), their advantage over MQ tends to be larger for larger systems. This is consistent with the lower performance observed for the smallest systems (Section 4.5.4.2). In contrast, the improvement of HMD^{+}_{25} over HMD, which isolates the contribution of the change-history data, shows no relevant correlation with size (-0.14). Since no system with more than 207 classes was evaluated, these results should not be extrapolated to larger systems.

Change localization in the developers' designs. The improvements show no relevant correlation with the SPCR (from -0.14 to 0.14) or the APC (from -0.13 to 0.10) of the

developers' designs. Thus, the gains in authoritativeness did not depend on how localized the changes were in the original designs.

Regarding the choice of threshold, lower values of p include more co-change relationships in the Enhanced HMD: HMD^{+}_{25} incorporates the 75% strongest relationships of the CRG, whereas HMD^{+}_{50} and HMD^{+}_{75} incorporate the strongest 50% and 25%, respectively. Among the three configurations, HMD^{+}_{25} achieved the lowest mean normalized edit distance to DEV (1.18), while HMD^{+}_{50} produced the number of packages closest to the developers' designs (a mean of six packages, equal to the developers' mean) and the highest single-package commits ratio (0.76); both obtained similar average numbers of packages per commit (1.39 and 1.40). However, the differences among the three thresholds were small and, for ZSS/N, not statistically significant (Table 4.13). Therefore, our results do not indicate a single best threshold, and its choice remains an open issue (Section 5.4).

4.5.6 Threats to Validity

Following the guidelines established by Wohlin et al. (Wohlin, 2012), we categorize the threats to the validity of this quantitative evaluation into construct, internal, external, and conclusion validity.

Construct Validity. These threats concern the relationship between the theory and the observations. A primary threat lies in the use of the developer-authored architecture (DEV) as the reference decomposition. DEV is not necessarily optimal: our analysis of outliers such as JGE (hyper-fragmented) and LJL (monolithic) suggests that human-authored designs may harbor architectural debt. Consequently, measuring algorithmic performance solely as the structural distance to DEV (via ZSS) might penalize decompositions that differ from DEV for legitimate reasons. To mitigate this, we supplemented ZSS with change-localization metrics (SPCR and APC), for which DEV serves as a baseline rather than a target. However, these metrics are computed on the same commit history used to build the *CRG*. Therefore, the advantage of the HMD^{+}_{p}

configurations over DEV on SPCR and APC is partly a consequence of their own objective and does not, by itself, show that the generated decompositions are better than those proposed by the developers. Such an assessment would require evaluating the decompositions on commits not used by the optimization and collecting the developers' opinions (Section 5.4). Additionally, the choice of the Jaccard coefficient to normalize commit intersections is a standard yet specific measure of set similarity. Alternative weighting mechanisms could potentially alter the graph's topology and the resulting HMD_p^+ configurations.

Internal Validity. Internal threats relate to factors that could influence the independent variables without the researchers' knowledge. A significant threat involves extracting and preprocessing the change history. Version control systems contain noise, such as large cross-cutting commits from automated refactoring tools, license updates, or file renames. While we mapped historical file paths to stable class identifiers to minimize data loss, large "transient" commits (as observed in JGE) can create artificial co-change dependencies, tricking the MDL fitness function into grouping unrelated domains. Furthermore, to ensure that the observed improvements were attributable to the change history data and not merely to expanding the search space into a hierarchical structure, we compared the HMD_p^+ variants against the purely structural HMD baseline, which shares the same representation, fitness function, and search algorithm. This isolation, however, is partial: we evaluated the change history data only within the hierarchical MDL-based formulation, and the comparison between MQ and HMD changes both the representation and the fitness function at once. Hence, we could not fully separate the individual contributions of the hierarchical representation, the Ψ metric, and the change history data, as well as possible interactions among them.

External Validity. External validity concerns the generalizability of the findings. The evaluation was conducted on 21 open-source systems. While these instances vary significantly in size (from 23 to 207 classes) and domain, they are all Java-based and developed in an open-source context. The change history patterns of commercial,

closed-source software with strict, centralized governance models might differ. Furthermore, the size of the evaluated sub-modules was intentionally bounded to approximately 200 classes. Because the HAC^* algorithm, coupled with the MDL fitness function, is computationally intensive, scaling this exact search procedure to large-scale legacy systems without further heuristic optimization remains a limitation of this study.

Conclusion Validity. Conclusion validity refers to the statistical significance and reliability of the inferences made. We employed the non-parametric Kruskal-Wallis test across the six configurations ($n = 126$ observations) to ensure our findings regarding the number of packages, SPCR, and APC were statistically rigorous ($p < 0.001$). For the absolute ZSS metric, the inherent exponential scaling of tree edit distance with system size initially masked statistical significance. We addressed this limitation by normalizing the data (ZSS/N) and applying paired tests (Friedman and Wilcoxon), which showed consistent improvements across the evaluated instances, ranging from 23 to 207 classes.

4.6 Developer-based Study

To complement the quantitative evaluation with a comparison against decompositions proposed by human subjects, we reused the dataset collected during the experimental study originally reported by Monçores *et al.* (2018). In that study, a group of software developers with varying levels of experience participated in an experiment focused on the distribution of software classes into packages.

The object of the study was JodaMoney[2], a Java library designed to store, manipulate, and represent monetary amounts. JodaMoney is a relatively small software project composed of 20 core classes. It was selected because its size and complexity were considered suitable for subjects to understand the relationships among classes and their respective responsibilities within the two-hour session allocated for the experiment.

In the original experiment, subjects were provided with a class diagram representing the classes of JodaMoney and were instructed to manually distribute these

[2] http://www.joda.org/joda-money/

classes into packages. The resulting distributions were subsequently compared with both the distribution proposed by the developers of JodaMoney and the distribution generated by a Large Neighborhood Search (LNS) optimization algorithm based on a flat formulation of the Software Module Clustering problem.

In the following sections, we compare the decompositions produced by the subjects, the developers, and the LNS algorithm with those generated by the proposed hierarchical optimization models (HMD^{+}_{25}, HMD^{+}_{50}, and HMD^{+}_{75}), as well as with the baseline algorithm (HMD). Parts of Sections 4.6.1 and 4.6.2 were adapted, with only minor modifications, from the paper reporting on the study (Monçores et al., 2018), since neither the characteristics of the subjects nor the clustering solutions they proposed changed in the present investigation.

Section 4.5 showed that the HMD^{+}_{p} configurations may over-fragment very small systems. The *JodaMoney* study complements that evaluation with a different kind of evidence: a comparison with decompositions proposed by human subjects for a system small enough to be understood within a two-hour session. Since it involves a single small system and data collected in a previous study, its results should be interpreted as complementary evidence, rather than as a validation of the practical usefulness of the approach.

4.6.1 Subject characterization

We invited twenty software developers with heterogeneous backgrounds in object-oriented programming and design to participate in the study. Of these, thirteen of these subjects agreed to take part in the experiment. The participants were either currently employed or had previous professional experience in the software development industry, including positions in both private and governmental companies. Most of these companies were based in Brazil, but one subject worked for a Canadian company, while a second one worked for a company based in the United States.

All participants filled out a characterization questionnaire designed to assess their level of experience. The objective of this questionnaire was to classify subjects as either

experienced or inexperienced developers, thereby enabling an analysis of whether subject experience influenced the study outcomes. To this end, we measured their programming and design skills by proposing two questions for each criterion. These questions would be answered on a three-point Likert scale representing low, medium, or high experience. On programming, we asked how many years the subject has worked as a programmer (up to 5 years, from 6 to 10 years, or more than 10 years) and about his/her experience with object-oriented programming (low, medium, or high). Concerning design, we asked about their experience in interpreting class diagrams (low, medium, or high) and experience with software architecture (low, medium, or high).

For each criterion, we observed whether there was agreement on the answers to both questions and assigned the agreed-upon experience level to the subject. An additional question was added to the characterization questionnaire to address academic background. Whenever disagreement arose on a given criterion, the answer to the question addressing academic background was used to resolve the dispute. We observed a broad range of academic qualifications among our subjects, though all had a Computer Science background. Two subjects were DSc students, three were MSc students, two completed a Master's degree program, two completed a *latu-sensu* post-graduate course, two completed graduation, and the last two were undergraduate students working as interns for their companies. Subjects holding a Master's degree were considered experienced, and if their answers to the questions addressing a given criterion were in disagreement, their classification drifted to the higher rank. On the other hand, undergraduate and graduate students were given a lower rank if a disagreement appeared in their assessments.

With respect to programming expertise, an agreement on the answers was reached for only 5 out of 13 subjects. Three subjects reported less than five years of programming practice and medium experience with object-oriented programming (OOP), being ranked as low-experienced programmers. Two subjects reported six to ten years of experience working as programmers and had high experience with OOP, and were ranked as highly experienced. An MSc student reported working as a programmer for more than ten years,

but had only medium experience with OOP and was ranked as a medium-experienced programmer. A subject holding an MSc degree reported working for six to ten years with OOP and was ranked as a highly experienced programmer. Finally, a DSc student reported working for less than five years with programming, but having high experience with OOP, and was ranked as highly experienced. We ended up with eight highly experienced, two medium-experienced, and three low-experienced programmers.

Regarding design expertise, an agreement on the answers was reached for 9 out of 13 subjects. An undergraduate subject reported medium experience in reading class diagrams and low experience in software architecture, and was ranked as a low-experienced designer. A subject holding an MSc degree reported low experience in interpreting class diagrams but high experience in software architecture and was ranked as a highly experienced designer. Another subject holding an MSc degree reported high experience in reading class diagrams, but medium experience in software architecture, and was ranked as a highly experienced designer. Finally, a DSc student reported medium experience in reading class diagrams and low experience in software architecture and was classified as a medium-experienced designer. In the end, we had four highly experienced, four medium-experienced, and five low-experienced designers.

Finally, subjects were classified as experienced or inexperienced according to the former criteria. Consistent with expectations, whenever a subject received a high score in design, he/she also received a high score in programming. Thus, highly experienced programmers with at least medium design experience were classified as experienced subjects, while the remaining ones were classified as inexperienced. Overall, we had seven experienced and six inexperienced subjects. The analyses presented in the next subsections consider these groups.

4.6.2 Solution characteristics

After completing the characterization questionnaire, the subjects received a class diagram depicting JodaMoney's core classes, their attributes, and relationships, along with a brief description of the project. We then asked the subjects to group the classes

into as many packages as they thought were necessary for the project. No further information was provided to the subjects, though they could search for more information on the Internet.

The solutions proposed by the subjects were all different. They were also different from the solutions provided by the optimization algorithms under analysis (HMD^{+}_{25}, HMD^{+}_{50}, HMD^{+}_{75}, HMD, and LNS) and JodaMoney's developers. LNS proposed distributing the classes into nine packages, with the largest having four classes and the smallest having a single one. HMD^{+}_{25} proposed a hierarchical distribution with two root packages: the first with six classes and the second with four classes and two packages, each with five classes. HMD^{+}_{50} produced a similar distribution, the first root package housing three classes and the second containing six classes and two child packages, one with five and the other with six classes. HMD^{+}_{75} followed its counterparts, with the first root package housing eight classes and the second containing two child packages: one with two classes and the other with two child packages with two and eight classes, respectively. HMD proposed the most complex design among the optimization algorithms, using three root packages (three, four, and five classes), the last with two child packages with six and two classes, respectively. JodaMoney's authors used only two packages, each holding ten classes.

The average number of packages proposed by the subjects was 6.08 (median=6). Experienced subjects proposed using, on average, 6.14 packages, while inexperienced ones proposed using six packages (the median was six for both groups). Two experienced subjects proposed the smallest solutions using three packages, while another experienced subject proposed the larger distribution using eleven packages. Experienced subjects used, on average, 2.0 single-class packages (median=2), while inexperienced ones used 1.2 such packages (median=1). The average difference between the largest package and the smallest one for experienced subjects was 4.7 classes (median=4), while for inexperienced subjects it was 5.2 classes (median=5). The largest package, comprised of 12 classes, was proposed by an experienced subject.

We observe a lack of clear agreement on how to distribute classes into packages, even for such a small project as JodaMoney. Both experienced and inexperienced subjects used more packages than expected from the literature, though the number of packages used by JodaMoney's developers is more consistent with such literature. Moreover, both groups of subjects used roughly the same number of single-class packages. Inexperienced subjects seemed slightly more inclined to the recommendation of avoiding such structures and not concentrating many classes in a single package.

Only five subjects proposed a hierarchical decomposition of classes into packages, while the eight remaining subjects opted for a flat distribution, with no package containing another package. Two out of these five subjects were experienced. This is an interesting aspect because providing a flat distribution of classes into packages is a known limitation of the most common formulation of the Software Module Clustering problem, and this limitation is addressed in the present work. While this does not seem to be a concern for many developers, at least for small projects, CRG variants and HMD can propose hierarchical distributions of classes into packages, and the baseline LNS provides only flat distributions.

4.6.3 Predictions on issue resolution

JodaMoney maintains its source code on the GitHub platform. The platform provides an issue-tracking system where bugs and requests for new features can be reported and monitored by the library's users as the development team works on them. GitHub also integrates the version-control system hosting the source code with the issue-tracking system. This integration enables developers to mark issues as resolved by referring to them in commit messages. Such *issue-related commits* allow researchers to identify the source-code modules modified to address a given issue.

As of May 2017, JodaMoney's issue-tracking system contained 50 registered issues, including 41 resolved and 9 open issues. Among the resolved issues, 23 were associated with commits to the version control system. Thirteen of these issues involved modifications to at least one Java file (a class among the core classes comprising

JodaMoney) that was not a test case. Six issues (#16, #24, #31, #35, #47, and #58) involved changes to two or more classes. The classes associated with these issues were used to determine whether the solutions found by the proposed optimization algorithms (HMD^{+}_{25}, HMD^{+}_{50}, and HMD^{+}_{75}) would be better predictors for classes related to the same issues than the solutions proposed by JodaMoney's authors, by the subjects, or by the baseline optimization algorithms (HMD and LNS).

All six issues involved classes assigned to the same package in the solution proposed by JodaMoney's developers. In contrast, only four of these issues involved classes grouped within the same package in the solution generated by the LNS algorithm. The *BigMoney* and *CurrencyUnit* classes were changed together to solve issue #31, but pertain to different packages in the solution found by LNS. Similarly, the *CurrencyUnit*, *CurrencyUnitDataProvider*, and *DefaultCurrencyUnitDataProvider* classes were changed together as part of issue #24's resolution, but are not part of the same package in the solution proposed by the algorithm. By comparison, five issues involve classes placed in the same package by HMD^{+}_{25}, HMD^{+}_{50}, and HMD. The exception for these algorithms was issue #31, which involved the *CurrencyUnit* and *BigMoney* classes. HMD^{+}_{75} produced a distribution that led to three single-package issues, still an improvement over LNS's results.

Experienced subjects proposed solutions that encompassed classes that were changed together to resolve, on average, 4.6 issues (median=5), while inexperienced subjects brought together classes that were changed to resolve, on average, 3.6 issues (median=4). Thus, for this small set of six issues, the solutions found by HMD^{+}_{25} and HMD^{+}_{50} grouped the classes involved in as many issues as *HMD* and the median experienced subject (five), and in more issues than *LNS* (four). Experienced developers, in their turn, perform better than inexperienced subjects and our baseline LNS.

4.6.4 Predictions on concomitant changes

JodaMoney has been developed for over eight years, from its first commit to the version control system (August 19, 2009) to the date when the data was collected for this

experiment. The log of the version control system records 282 commits in this period, 255 of which were performed by the most active developer engaged in the project (Stephen Colebourne). After him, one developer performed three commits, six developers performed two commits, and twelve developers performed a single commit.

Half of the 282 commits involve one or more Java classes, and 129 of them involve at least one Java class that is not a test case. Sixty commits involve two or more classes among the 20 core classes in JodaMoney's design. The average commit among the sixty that were analyzed involved 4.3 classes (median=3), and the largest commit involved all 20 core classes.

We define a *single-package commit* as a commit to the version control system involving two or more classes so that all these classes reside in the same package under a given distribution of classes to packages. A distribution having more single-package commits than a second distribution is considered a better distribution of classes to packages because it places classes that change together frequently in the same package.

We investigated whether the distributions produced by the proposed optimization algorithms (HMD^{+}_{25}, HMD^{+}_{50}, and HMD^{+}_{75}) would outperform the distribution proposed by JodaMoney's authors, by the subjects, and by the baseline algorithms (HMD and LNS). Table 4.17 reports our findings. Its first column presents the source of the distribution, the second presents the number of single-package commits in the distribution, and the third presents this number as a percentage of the sixty commits under analysis.

We observe that the distribution of classes to packages proposed by JodaMoney's authors outperforms all optimization algorithms and developers. On the other hand, the proposed algorithms led to more single-package commits than the baselines, particularly the flat LNS, and HMD^{+}_{50} matched the median of the experienced subjects. HMD^{+}_{50} performs slightly better than HMD^{+}_{25} and HMD^{+}_{75}, with its solution leading to three extra single-package commits. The baseline algorithms produce results comparable to those of inexperienced developers.

Table 4.17: Performance of the distribution of classes to packages proposed by JodaMoney's authors, the distributions proposed by the optimization algorithms, and the distributions proposed by the subjects as of the number of single-package commits.

Distribution Source	Single-package Commits	
	Count	Percentage
JodaMoney's authors	46	76.7%
HMD^{+}_{25}	28	46.7%
HMD^{+}_{50}	31	51.7%
HMD^{+}_{75}	28	46.7%
HMD	24	40.0%
LNS	19	31.7%
Experienced subjects (median)	31	51.7%
Inexperienced subjects (median)	24	40.0%

4.6.5 Comparison among solutions

Given that the solution proposed by JodaMoney's authors was the best predictor for concomitant changes and issue-related changes, next, we examine how close the solutions found by optimization and proposed by our subjects are to the one used by JodaMoney's authors using the Tree Distance metric described in Section 4.5.1. This metric counts the minimum number of node insertions, deletions, and relabelings required to transform one tree into another, allowing us to measure how closely a hierarchical distribution of classes to packages is to another. Smaller tree distances represent closer distributions, while larger distances represent distributions that are very different from one to another. In this sense, flat distributions are treated as one-level hierarchical distributions.

Table 4.18 presents the tree distance values for the solution found by optimization and those proposed by our subjects (median tree distance) when compared to the solution proposed by JodaMoney's authors. We observe that HMD^{+}_{25} and HMD^{+}_{50} produced the solutions closest to the one proposed by JodaMoney's authors, closer than those of the remaining optimization algorithms and than the median solutions of both

Table 4.18: Tree distance between the distribution of classes to packages by the optimization algorithms and subjects to the distribution proposed by JodaMoney's authors.

Distribution Source	Tree Distance
HMD^{+}_{25}	22
HMD^{+}_{50}	22
HMD^{+}_{75}	29
HMD	28
LNS	32
Experienced subjects (median)	26
Inexperienced subjects (median)	26.5

groups of subjects, whereas HMD^{+}_{75} was slightly farther than *HMD*. Thus, if we assume that the clustering provided by the authors is a good solution – given the experience of the project's most active developer and its ability to predict classes changed together – these results suggest that HMD^{+}_{25} and HMD^{+}_{50} improve over the baseline algorithms for this project. Whether such solutions would actually help developers organize their systems, however, cannot be concluded from this study (Section 4.6.7).

4.6.6 Discussion

The data presented in the former subsections suggest that, for the *JodaMoney* project, HMD^{+}_{25} and HMD^{+}_{50} improve over the single-objective algorithms used as baselines for hierarchical (*HMD*) and flat (*LNS*) clustering: they placed classes that changed together in the same commits in the same package more often than these algorithms, and did so for issue-related changes at least as often. The solution proposed by JodaMoney's authors, on the other hand, remains the best predictor for both kinds of changes.

Beyond the raw performance metrics, the difference observed between the purely structural approaches (*HMD* and *LNS*) and the hybrid formulations (HMD^{+}_{25} and HMD^{+}_{50}) is consistent with the view that static code structure offers only a partial view of

how classes relate to each other. By incorporating version control data through the *CRG*, the optimization process gains access to relationships that reflect how developers have actually modified the system, which favors grouping classes that are frequently changed together.

Nevertheless, the original human-authored architecture remains the best predictor for localized changes, revealing the limits of automated optimization. Human designers do not build software solely on structural coupling or history; they also rely on domain knowledge, abstraction, and semantics that static code analysis does not capture. The fact that HMD^{+}_{25} and HMD^{+}_{50} bridged part of this gap suggests that version control history can capture some of this design knowledge. These results, however, should be interpreted with caution: they come from a single small system, and the HMD^{+}_{p} configurations used JodaMoney's change history – which includes the commits and issues analyzed in this study – whereas the subjects and the baseline algorithms had no access to it.

4.6.7 Threats to Validity

The study reported in this section has some limitations that may threaten the validity of its results. First, the study is based on a single software project used to evaluate the usefulness of solutions to the Software Module Clustering problem that were found through different means. JodaMoney is a small software library comprising 20 classes and may not be representative of the properties of the average software project. However, the project size is reasonable, given that subjects were not expected to invest more than two hours in the study. Moreover, we were able to gather enough information to discuss the limitations of Software Module Clustering, even from this small project.

Besides being small, JodaMoney had 50 issues registered in its issue-tracking system and 282 commits registered in its version control system by the time we collected information for the study. The number of issues related to two or more classes was eventually reduced to six, and the number of commits related to two or more classes was

reduced to sixty. These are relatively small numbers for the average software project and may be biased towards the design chosen by JodaMoney's developers.

A third limitation refers to the number of subjects participating in the study. While only thirteen subjects participated in the study, they were all active software developers working for companies in Brazil or abroad. Therefore, we had a small but diversified sample of subjects.

Finally, many subjects requested more information after completing the experimental analysis. Seven subjects asked for more information about the classes, stating that the role played by some of these classes could not be discerned by the information available in the class diagram they received – their names, attributes, and relationships. Two subjects asked for sequence diagrams, which capture dynamic relationships among classes as they call each other's methods to perform some task. One subject asked for a description of the requirements of the library under analysis. Neither of these pieces of information was readily available, and the subjects had to work with limited information, while the data they requested would probably be available in a real project setting. We cannot rule out the possibility that their answers might be different if they were given such data.

4.7 Summary

This chapter introduced and empirically validated the core contribution of this thesis: the use of co-change relationships between classes to guide search-based hierarchical software clustering. By recognizing that structural dependencies capture only the mechanical skeleton of a system, we proposed the Commit Relation Graph (CRG) to map the conceptual, implicit relationships that drive actual software maintenance.

Addressing RQ2, our findings indicate that incorporating data from the software's change history to capture implicit class relationships improves the quality of hierarchical software decompositions. The quantitative evaluation, conducted across 21 real-world open-source systems, demonstrated that augmenting the Minimum

Description Length (MDL) fitness function with CRG-derived dependencies – formalized as the HMD_p^+ model – effectively mitigates over-fragmentation inherent in purely structural baselines. The HMD_p^+ configurations successfully bridged the gap between static code representation and dynamic developer intent. Specifically, the HMD_{25}^+ configuration improved the descriptive authoritativeness of the generated architectures, reducing the normalized Tree Edit Distance (ZSS/N) to the developers' original design by 14% relative to the purely structural hierarchical baseline (*HMD*), a comparison that isolates the contribution of the change-history data, and by 26% relative to the non-hierarchical baseline (*MQ*). Furthermore, on the analyzed commit history, the HMD_{50}^+ configuration achieved single-package commit ratios (SPCR) and average numbers of packages affected per commit (APC) similar, on average, to those of the original human-authored designs.

A complementary study, based on data from a previous experiment with human subjects on the *JodaMoney* project (Monçores et al., 2018), indicated that, for this system, the decompositions generated by HMD_{25}^+ and HMD_{50}^+ placed classes that changed together in the same package more often than the flat and hierarchical baselines, with results comparable to those of the experienced subjects. As this evidence comes from a single small system, it complements, but does not establish, the practical usefulness of the approach.

Ultimately, the findings of this chapter reinforce that software architecture is not a static artifact, but a structure that evolves over time. The HMD_p^+ approach translates version control history into additional dependencies that bring the generated decompositions closer to those proposed by developers. Whether it can help developers analyze and restructure real systems, particularly large ones, remains to be investigated.

5 CONCLUSIONS

This chapter concludes the thesis by summarizing the research findings, addressing the proposed research questions, acknowledging its limitations, and suggesting avenues for future research.

5.1 Overview

The Software Module Clustering (SMC) problem has historically been approached as a non-hierarchical graph partitioning challenge. While traditional optimization techniques driven by structural metrics, such as Modularization Quality (*MQ*), succeed in isolating highly coupled components, they frequently produce overly fragmented partitions. These flat architectures consistently fail to capture the nested, conceptual organization intended by software developers, creating a persistent gap between automated structural clustering and real-world semantic design.

This thesis proposed a hybrid, search-based hierarchical approach to address this gap. By combining the Minimum Description Length (*MDL*) principle with change history data extracted from version control systems – formalized as the Commit Relation Graph (*CRG*) – we established a methodology that respects both the physical dependencies of the source code and the implicit, historical relationships that dictate software maintenance. Through experimental evaluations across multiple open-source systems, we found evidence that co-change relationships – classes frequently modified together in the same commits – complement static structural dependencies and bring the generated hierarchical decompositions closer to those proposed by developers, indicating that the change history is a relevant source of information for the SMC problem.

5.2 Addressing the Research Questions

The primary objective of this research was to investigate whether hierarchical representations and change history data could improve automated software clustering. This objective was divided into two core research questions, which are addressed below based on the empirical evidence gathered.

RQ1: Does using a hierarchical-based representation and metric to guide the search for solutions to the SMC problem improve the generated solutions compared to non-hierarchical approaches?

Yes. The experimental study presented in Chapter 3 showed that employing the *MDL* metric to optimize a Hierarchical Modular Decomposition (*HMD*) improves the authoritativeness of the generated solutions over the flat *MQ* baseline. The *MQ* optimization exhibited a systemic bias toward over-fragmentation, producing an average of 56.6 packages compared to 12.8 in *DEV* and 30.3 in the *MDL* solutions, which achieved higher *MoJoFM* values in nine of the ten instances. The same comparison, repeated on 21 sub-modules in Chapter 4, confirmed this pattern (averages of 53, 6, and 21 packages for *MQ*, *DEV*, and *HMD*, respectively), with statistically significant differences in the normalized tree edit distance to the developers' designs. While the structural *HMD* approach produced a scale closer to that of the developers', it still left a semantic gap, indicating that static code dependencies alone are often too granular to fully recover high-level architectural groupings.

RQ2: Does incorporating data from the software's change history to capture implicit relationships between classes improve the quality of hierarchical software decompositions?

Yes, within the hierarchical MDL-based formulation evaluated in this thesis. Integrating the Commit Relation Graph (*CRG*) into the MDL-based optimization, formalized as the HMD_p^+ configurations in Chapter 4, reduced the identified semantic gap and improved the authoritativeness of the generated architectures. By penalizing the

separation of classes that frequently change together, the HMD_p^+ configurations reduced the normalized Tree Edit Distance (ZSS/N) by 11% to 14% relative to the purely structural HMD – the comparison that isolates the contribution of the change-history data –, with statistically significant differences for all three thresholds. Relative to the non-hierarchical baseline, the reduction reached 26% (HMD_{25}^+), combining the effects of the hierarchical formulation and the change-history data. Furthermore, on the analyzed commit history, the HMD_{50}^+ configuration yielded SPCR and APC values similar, on average, to those of the original human-authored designs. Among the three thresholds, HMD_{25}^+ produced the decompositions closest to the developers' designs, whereas HMD_{50}^+ produced the package count closest to theirs and the highest SPCR; since these differences were small and not statistically significant for ZSS/N, the choice of threshold remains open.

5.3 Limitations

While the proposed methodology demonstrates improvements, the experimental evaluation has limitations and exposed boundary conditions where the approach requires careful application:

- **Vulnerability in Micro-architectures:** The *MDL* fitness function is mathematically driven to seek hierarchical depth and compression. In highly specialized atomic libraries with few packages (e.g., the LJJ submodule, with 23 classes across 2 packages), the algorithm tends to over-fragment cohesive domains. Forcing change-driven hierarchical clustering onto legitimately atomic micro-libraries introduces unnecessary structural friction;

- **Sensitivity to Transient Global Refactorings:** The construction of the *CRG* assumes that co-modifications indicate a conceptual relationship. However, massive cross-cutting commits, such as automated license updates or global method-renaming tasks, create artificial co-change dependencies (*e.g.*, the "Shotgun Surgery" anti-pattern). While the threshold mechanism filters out minor

noise, hyper-fragmented legacy systems with dispersed commit histories can still affect the algorithm's capability to identify true semantic boundaries;

- **Size of the evaluated systems:** The evaluation was restricted to sub-modules with up to about 200 classes because of the computational cost of the HAC^* algorithm combined with the *MDL* fitness function. Neither the feasibility of the approach nor its effects on large systems, with thousands of classes, were assessed;

- **Practical usefulness:** The generated decompositions were compared with the developers' designs and with the projects' change history, but we did not assess whether they would help developers understand or maintain the analyzed systems;

5.4 Future Work

The findings of this thesis open several avenues for future research in Search-Based Software Engineering and architecture recovery:

- **Co-change relationships in other Software Engineering problems:** The results reported in Chapter 4 indicate that relationships derived from classes frequently modified together in the same commits complement a system's static structural dependencies and bring the generated decompositions closer to those proposed by its developers. This suggests that this information may also be useful for other problems whose formulations rely mostly on structural data. Future research could investigate its use in problems such as search-based refactoring (*e.g.*, recommending move method or move class operations), the class responsibility assignment problem (Bowman et al., 2010), and regression test selection and prioritization, incorporating the *CRG* either as a complementary set of dependencies, as done in the HMD_p^+ representation, or as an additional objective in multi-objective formulations;

- **Alternative co-change measures:** The *CRG* measures the strength of co-change relationships through the Jaccard coefficient, which captures only the symmetric

co-occurrence of classes in commits. However, co-change coupling may be asymmetric: a class A may be changed almost always together with a class B, while B also changes with many other classes, so that A is more coupled to B than B is to A. Measures from association rule mining could be explored to weight and select these relationships, such as the support and confidence of association rules (Agrawal et al., 1993; Zimmermann; Zeller, et al., 2005) and other interestingness measures (Tan et al., 2004). Since confidence is directional, it could yield directed change-related edges, whereas minimum support thresholds could discard relationships backed by few commits;

- **Separating the contributions of representation, metric, and change history:** In this thesis, we evaluated the change-history data only within the hierarchical MDL-based formulation, and the comparison with the flat baseline changed the representation and the fitness function at once. We could isolate them and evaluate the CRG-derived edges in a flat formulation (*e.g.*, adding them to the *MDG* optimized by *MQ*) and the Ψ metric restricted to flat partitions, quantifying the individual contribution of each component and their interactions;

- **Additional similarity measures:** The comparison with the developers' decompositions relied on MoJoFM (Chapter 3) and on a tree edit distance (Chapter 4). Other measures could complement this analysis, such as the precision and recall of pairs of classes placed in the same package (Anquetil et al., 1999), measures based on an optimal matching between the packages of two decompositions, EdgeSim and MeCl, which also take the dependencies between classes into account (Mitchell; Mancoridis, 2001), and frameworks designed for nested decompositions (Shtern et al., 2004);

- **Integration of Natural Language Processing (NLP):** As observed during the qualitative analysis, developers frequently group classes based on naming conventions and domain semantics (*e.g.*, classes sharing a `Parameter` suffix). While initial studies by Rodrigues (Rodrigues, 2025) explored word embeddings

for Java software remodularization, his approach focused primarily on static and lexical features without incorporating version-control commit history. Therefore, integrating NLP –specifically leveraging code-aware word embeddings rather than general-purpose text embeddings – to extract semantic similarity from class names, variables, and comments could serve as a third dimension of the dependency graph, unifying structure, history, and semantics into a single optimization process;

- **Dynamic Threshold Calculation:** The current HMD_p^+ model relies on fixed percentiles ($p \in \{25, 50, 75\}$). Future research could investigate dynamic threshold mechanisms that automatically determine the optimal *CRG* cut-off based on the target system's specific size, density, and historical volatility;

- **Tooling and IDE Integration:** To maximize industry impact, the optimization algorithm could be adapted into a continuous integration tool or an IDE plugin. By evaluating the complexity of a software project (Ψ) in real time, such a tool could warn developers about architectural drift during code reviews, preventing the degradation of modular boundaries before they are committed to the repository;

- **Algorithmic Scalability and Heuristics**: While the HAC^* algorithm coupled with the Ψ fitness function was evaluated on instances of up to about 200 classes, applying this search procedure to large systems (with thousands of classes) requires further heuristic optimizations or parallelization strategies to keep computation time reasonable, as well as studies assessing whether the gains observed here also hold for such systems;

- **Evaluation with Random Graph Instances:** Generating random graph instances to systematically compare the clustering behavior and the resulting module sizes of the MQ and MDL metrics under controlled scenarios;

- **Machine Learning for Cost Function Design:** Exploring computational learning models, such as Reinforcement Learning algorithms (e.g., PPO or Q-learning), to

automatically learn and map the cost function based on the authoritative decompositions provided by software developers;

- **Alternative Clustering Paradigms:** Investigating the applicability of alternative clustering techniques, such as spatial clustering and advanced graph clustering algorithms, to further bypass structural constraints and improve partitioning accuracy;

- **Large Language Models (LLMs):** LLMs could be explored in two complementary ways. First, as a baseline, by asking a model to propose a decomposition of the classes into packages – an inexpensive alternative that captures the textual dimension of the relationships among classes (Section 2.6). Second, use the model as an information source for optimization by estimating semantic similarity between class pairs, which could be added to the dependencies evaluated by Ψ, as done with the *CRG*, while keeping the search itself outside the model. In both cases, the evaluation must control for the risk that the model has seen the analyzed systems during training, for instance, by using proprietary systems or systems created after the model's training cutoff;

- **Information Gain Heuristics:** Incorporating concepts such as information gain – commonly used in decision tree training methods – to refine the heuristic criteria for merging or splitting clusters during the optimization process;

- **Predictive Risk and Failure Analysis:** Utilizing the Commit Relation Graph (CRG) not only for clustering but also as an analytical tool to identify anomalous architectural groupings, fragile code segments, and classes with a higher propensity for failure or high-risk maintenance;

- **Microservices Identification and Migration:** Extending the practical application of the co-modification-based clustering (CRG) to serve as a direct diagnostic tool for architecture teams looking to refactor and migrate legacy monolithic systems toward microservices architectures;

- **Multi-objective formulation:** Instead of merging structural and co-change dependencies into a single graph controlled by a fixed threshold, the Ψ values computed over each type of dependency could be treated as separate objectives, allowing the search to present developers with a set of trade-offs between structural and change-history criteria;

- **Temporal validation:** In this thesis, the CRG and the change-localization metrics (SPCR and APC) were computed on the same commit history. A temporal split – building the CRG with the commits up to a given date and computing SPCR and APC on the subsequent commits – would assess whether the generated decompositions also localize future changes, avoiding the overlap between the data used for optimization and for evaluation;

- **Evaluation with developers:** The decompositions generated by the proposed approach were not assessed by developers. Future studies could present them to the developers of the analyzed systems or to development teams in companies, collecting their opinion on whether the proposed packages are meaningful and useful for understanding and maintaining the systems;

- **Preprocessing of commits:** The *CRG* considers each commit as a separate transaction and does not filter commits by size or purpose. Future work could analyze how common large commits are in each project and discard those exceeding a threshold derived from the distribution of commit sizes, group commits performed within short time windows or related to the same task, and use commit messages – possibly classified with the help of LLMs – to identify commits that are less relevant for modularization, such as formatting changes, license updates, or global renamings;

- **Robustness and instance characteristics:** The stability of the proposed approach could be assessed by applying small perturbations to the instances – for instance, removing or adding dependencies and commits –, which would also allow

estimating confidence intervals for the results. In addition, characterizing the instances in terms of structural properties (e.g., density of dependencies) and evolutionary properties (e.g., number of commits, number of developers, project age, and commit frequency) could reveal which characteristics favor the use of change history;

- **Other domains:** The proposed approach does not depend on specific properties of source code: it requires only a set of structural dependencies among elements and a history of joint modifications of these elements. It could therefore be applied to other domains with the same characteristics, such as relational databases, whose tables are related by foreign keys and are often changed together by the same migrations, or legal codes, whose articles reference each other and are amended together over time.

REFERENCES

AGRAWAL, R.; IMIELIŃSKI, T.; SWAMI, A. Mining association rules between sets of items in large databases. *In:* PROCEEDINGS of the 1993 ACM SIGMOD International Conference on Management of Data. [*S. l.: s. n.*], 1993. p. 207–216. DOI: 10.1145/170035.170072.

AGUILAR-RUIZ, J. S.; RAMOS, I.; SANTOS, J. C. R.; TORO, M. An evolutionary approach to estimating software development projects. **Information and Software Technology**, v. 43, n. 14, p. 875–882, 2001. Publisher: Elsevier. DOI: 10 . 1016 / S0950 - 5849(01) 00193 - 8. Available from: https://doi.org/10.1016/S0950-5849(01)00193-8.

AMOUI, M.; MIRARAB, S.; ANSARI, S.; LUCAS, C. A genetic algorithm approach to design evolution using design pattern transformation. *In:* available from: https://api.semanticscholar.org/CorpusID:2812518.

ANQUETIL, N.; LETHBRIDGE, T. Experiments with Clustering as a Software Remodularization Method. *In:* SIXTH Working Conference on Reverse Engineering, WCRE '99, Atlanta, Georgia, USA, October 6-8, 1999. [*S. l.*]: IEEE Computer Society, 1999. p. 235–255. DOI: 10 . 1109 / WCRE . 1999 . 806964. Available from: https://doi.org/10.1109/WCRE.1999.806964.

ANTONIOL, G.; PENTA, M. D.; HARMAN, M. Search-Based Techniques Applied to Optimization of Project Planning for a Massive Maintenance Project. *In:* PROCEEDINGS of the 21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary. [*S. l.*]: IEEE Computer Society, 2005. p. 240–249. DOI: 10 . 1109 / ICSM . 2005 . 79. Available from: https://doi.org/10.1109/ICSM.2005.79.

ARCURI, A.; BRIAND, L. C. A practical guide for using statistical tests to assess randomized algorithms in software engineering. *In:* PROCEEDINGS of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011. [*S. l.*]: ACM, 2011. p. 1–10. DOI: 10 . 1145 / 1985793 . 1985795. Available from: https://doi.org/10.1145/1985793.1985795.

BAGNALL, A. J.; RAYWARD-SMITH, V. J.; WHITTLEY, I. M. The next release problem. **Information and Software Technology**, v. 43, n. 14, p. 883–890, 2001. ISSN 0950-5849. DOI: 10.1016/S0950-5849(01)00194-X. Available from: https://doi.org/10.1016/S0950-5849(01)00194-X.

BARROS, M. d. O. An analysis of the effects of composite objectives in multiobjective software module clustering. *In:* PROCEEDINGS of the 14th Annual Conference on Genetic and Evolutionary Computation (GECCO '12). [*S. l.*]: ACM, 2012. p. 1205–1212. DOI: 10.1145/2330163.2330330.

BARROS, M. d. O. An experimental evaluation of the importance of randomness in hill climbing searches applied to software engineering problems. **Empirical Software Engineering**, v. 19, n. 5, p. 1423–1465, 2014. DOI: 10.1007/S10664-013-9294-4. Available from: https://doi.org/10.1007/s10664-013-9294-4.

BARROS, M. d. O.; FARZAT, F. d. A.; TRAVASSOS, G. H. Learning from optimization: A case study with Apache Ant. **Information and Software Technology**, v. 57, p. 684–704, 2015. DOI: 10.1016/J.INFSOF.2014.07.015. Available from: https://doi.org/10.1016/j.infsof.2014.07.015.

BATE, I.; POULDING, S. M. Editorial for the special issue on search-based software engineering. **Software: Practice and Experience**, v. 41, n. 5, p. 467–468, 2011. DOI: 10.1002/SPE.1056. Available from: https://doi.org/10.1002/spe.1056.

BAVOTA, G.; DE LUCIA, A.; MARCUS, A.; OLIVETO, R. Using structural and semantic measures to improve software modularization. **Empirical Software Engineering**, v. 18, n. 5, p. 901–932, 2013. DOI: 10.1007/s10664-012-9226-8.

BAVOTA, G.; GETHERS, M.; OLIVETO, R.; POSHYVANYK, D.; LUCIA, A. d. Improving software modularization via automated analysis of latent topics and dependencies. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, ACM New York, NY, USA, v. 23, n. 1, p. 1–33, 2014.

BECK, F.; DIEHL, S. On the congruence of modularity and code coupling. *In:* PROCEEDINGS of the 19th ACM SIGSOFT Symposium and the 13th European

Conference on Foundations of Software Engineering (ESEC/FSE). [*S. l.: s. n.*], 2011. p. 354–364. DOI: 10.1145/2025113.2025162.

BECK, F.; DIEHL, S. On the impact of software evolution on software clustering. **Empirical Software Engineering**, v. 18, n. 5, p. 970–1004, 2013. DOI: 10.1007/s10664-012-9225-9.

BOTELHO, A. L. V.; SEMAAN, G. S.; OCHI, L. S. Agrupamento de Sistemas Orientados a Objetos com Metaheurísticas Evolutivas. *In:* ANAIS do VII Simpósio Brasileiro de Sistemas de Informação (SBSI 2011). Brasil: Sociedade Brasileira de Computação, May 2011. p. 153–165. DOI: 10.5753/sbsi.2011.14573. Available from: https://sol.sbc.org.br/index.php/sbsi/article/view/14573. Visited on: 18 Feb. 2024.

BOWMAN, M.; BRIAND, L. C.; LABICHE, Y. Solving the Class Responsibility Assignment Problem in Object-Oriented Analysis with Multi-Objective Genetic Algorithms. **IEEE Transactions on Software Engineering**, v. 36, n. 6, p. 817–837, 2010. DOI: 10.1109/TSE.2010.70.

BRIAND, L. C.; MORASCA, S.; BASILI, V. R. Defining and Validating Measures for Object-Based High-Level Design. **IEEE Transactions on Software Engineering**, v. 25, p. 722–743, 1999. Available from: https://api.semanticscholar.org/CorpusID:6662624.

BROWN, W. J.; MALVEAU, R. C.; MCCORMICK, H. W.; MOWBRAY, T. J. **AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis**. New York, NY, USA: John Wiley & Sons, Inc., 1998.

BURGESS, C. J.; LEFLEY, M. Can genetic programming improve software effort estimation? A comparative evaluation. **Information and Software Technology**, v. 43, p. 863–873, 2001. Available from: https://api.semanticscholar.org/CorpusID:206106375.

CLAUSET, A.; NEWMAN, M. E. J.; MOORE, C. Finding community structure in very large networks. en. **Physical Review E**, v. 70, n. 6, p. 066111, Dec. 2004. ISSN 1539-3755, 1550-2376. DOI: 10.1103/PhysRevE.70.066111. Available from:

https://link.aps.org/doi/10.1103/PhysRevE.70.066111. Visited on: 4 Dec. 2023.

COHEN, J. A power primer. eng. **Psychological Bulletin**, v. 112, n. 1, p. 155–159, July 1992. ISSN 0033-2909. DOI: 10.1037//0033-2909.112.1.155.

COLLINS, R. J.; JEFFERSON, D. R. Selection in Massively Parallel Genetic Algorithms. *In:* BELEW, R. K.; BOOKER, L. B. (eds.). **Proceedings of the 4th International Conference on Genetic Algorithms, San Diego, CA, USA, July 1991**. [*S. l.*]: Morgan Kaufmann, 1991. p. 249–256.

CORAZZA, A.; DI MARTINO, S.; MAGGIO, V.; SCANNIELLO, G. Weighing lexical information for software clustering in the context of architecture recovery. **Empirical Software Engineering**, v. 21, n. 1, p. 72–103, 2016. DOI: 10.1007/s10664-014-9347-3.

CORMACK, R. M. A Review of Classification. **Journal of the Royal Statistical Society. Series A (General)**, v. 134, n. 3, p. 321–367, 1971. Publisher: [Royal Statistical Society, Wiley]. ISSN 0035-9238. DOI: 10.2307/2344237. Available from: https://www.jstor.org/stable/2344237. Visited on: 26 Dec. 2023.

D'AMBROS, M.; LANZA, M.; ROBBES, R. On the Relationship Between Change Coupling and Software Defects. *In:* PROCEEDINGS of the 16th Working Conference on Reverse Engineering (WCRE). [*S. l.: s. n.*], 2009. p. 135–144. DOI: 10.1109/WCRE.2009.19.

DIAS, C.; OCHI, L. Efficient evolutionary algorithms for the clustering problem in directed graphs. *In:* THE 2003 Congress on Evolutionary Computation, 2003. CEC '03. [*S. l.: s. n.*], Dec. 2003. v. 2, 983–990 vol.2. DOI: 10.1109/CEC.2003.1299774. Available from: https://ieeexplore.ieee.org/document/1299774. Visited on: 18 Feb. 2024.

DOVAL, D.; MANCORIDIS, S.; MITCHELL, B. Automatic clustering of software systems using a genetic algorithm. *In:* STEP '99. Proceedings Ninth International Workshop Software Technology and Engineering Practice. Pittsburgh, PA, USA: IEEE Comput. Soc, 1999. p. 73–81. ISBN 978-0-7695-0328-8. DOI:

10 . 1109 / STEP . 1999 . 798481. Available from: http://ieeexplore.ieee.org/document/798481/. Visited on: 12 Feb. 2024.

DURILLO, J. J.; ZHANG, Y.; ALBA, E.; HARMAN, M.; NEBRO, A. J. A Study of the Bi-Objective next Release Problem. **Empirical Software Engineering**, v. 16, n. 1, p. 29–60, Feb. 2011. Place: USA Publisher: Kluwer Academic Publishers. ISSN 1382-3256. DOI: 10.1007/s10664-010-9147-3. Available from: https://doi.org/10.1007/s10664-010-9147-3.

FALKENAUER, E. A hybrid grouping genetic algorithm for bin packing. en. **Journal of Heuristics**, v. 2, n. 1, p. 5–30, June 1996. ISSN 1572-9397. DOI: 10.1007/BF00226291. Available from: https://doi.org/10.1007/BF00226291. Visited on: 18 Feb. 2024.

FOWLER, M. **Refactoring: Improving the Design of Existing Code**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

FRASER, G.; SOUZA, J. T. d. Guest editorial: search-based software engineering. **Empirical Software Engineering**, p. 1–3, 2018. Available from: https://api.semanticscholar.org/CorpusID:14630566.

GALL, H.; HAJEK, K.; JAZAYERI, M. Detection of Logical Coupling Based on Product Release History. *In:* PROCEEDINGS of the International Conference on Software Maintenance (ICSM). [*S. l.: s. n.*], 1998. p. 190–198. DOI: 10.1109/ICSM.1998.738508.

GALL, H.; JAZAYERI, M.; KRAJEWSKI, J. CVS release history data for detecting logical couplings. *In:* PROCEEDINGS of the 6th International Workshop on Principles of Software Evolution (IWPSE). [*S. l.: s. n.*], 2003. p. 13–23. DOI: 10.1109/IWPSE.2003.1231205.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design patterns: elements of reusable object-oriented software**. [*S. l.*]: Pearson Deutschland GmbH, 1995.

GAREY, M. R.; JOHNSON, D. S. **Computers and Intractability; A Guide to the Theory of NP-Completeness**. USA: W. H. Freeman & Co., 1979. ISBN 0-7167-1045-5.

GIBBS, S. J.; TSICHRITZIS, D.; CASAIS, E.; NIERSTRASZ, O.; PINTADO, X. Class management for software communities. **Communications of the ACM**, v. 33, p. 90–103, 1990. Available from: https://api.semanticscholar.org/CorpusID:16220359.

GLOVER, F.; KOCHENBERGER, G. A.; HILLIER, F. S. (eds.). **Handbook of Metaheuristics**. Boston, MA: Springer US, 2003. v. 57. (International Series in Operations Research & Management Science). ISBN 978-1-4020-7263-5 978-0-306-48056-0. DOI: 10.1007/b101874. Available from: http://link.springer.com/10.1007/b101874. Visited on: 18 Feb. 2024.

GORDON, A. D. A Review of Hierarchical Classification. **Journal of the Royal Statistical Society. Series A (General)**, v. 150, n. 2, p. 119–137, 1987. Publisher: [Royal Statistical Society, Wiley]. ISSN 0035-9238. DOI: 10.2307/2981629. Available from: https://www.jstor.org/stable/2981629. Visited on: 10 Jan. 2024.

GOSLING, J.; JOY, B.; STEELE, G.; BRACHA, G.; BUCKLEY, A.; SMITH, D.; BIERMAN, G. **The Java Language Specification, Java SE 21 Edition**. [*S. l.*]: Oracle America, Inc., 2023. Available from: https://docs.oracle.com/javase/specs/jls/se21/html/jls-7.html.

HALL, M.; MCMINN, P. An analysis of the performance of the bunch modularisation algorithm's hierarchy generation approach. *In:* 4 th Symposium on Search Based-Software Engineering. [*S. l.: s. n.*], 2012. p. 19.

HARMAN, M.; HIERONS, R.; PROCTOR, M. A new representation and crossover operator for search-based optimization of software modularization. *In:* PROCEEDINGS of the 4th Annual Conference on Genetic and Evolutionary Computation. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., July 2002. (GECCO'02), p. 1351–1358. ISBN 978-1-55860-878-8. Visited on: 11 Feb. 2024.

HARMAN, M.; JONES, B. F. Search-based software engineering. **Information and Software Technology**, v. 43, n. 14, p. 833–839, 2001. DOI: 10.1016/S0950-5849(01)00189-6. Available from: https://doi.org/10.1016/S0950-5849(01)00189-6.

HARMAN, M.; SWIFT, S.; MAHDAVI, K. An empirical study of the robustness of two module clustering fitness functions. *In:* ANNUAL Conference on Genetic and Evolutionary Computation. [*S. l.: s. n.*], 2005. Available from: https://api.semanticscholar.org/CorpusID:12653212.

JONES, D. S. **Elementary Information Theory**. [*S. l.*]: Oxford University Press, Aug. 1979.

KLEINBERG, J.; TARDOS, E. **Algorithm design**. [*S. l.*]: Pearson Addison Wesley, 2006. ISBN 0-321-29535-8.

KOZA, J. R. **Genetic programming - on the programming of computers by means of natural selection**. [*S. l.*]: MIT Press, 1993. (Complex adaptive systems). ISBN 978-0-262-11170-6.

KUHN, A.; DUCASSE, S.; GÎRBA, T. Semantic clustering: Identifying topics in source code. **Information and Software Technology**, v. 49, n. 3, p. 230–243, 2007. DOI: 10.1016/j.infsof.2006.10.017.

LARMAN, C. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process. *In:* available from: https://api.semanticscholar.org/CorpusID:119922699.

LAWLER, E. L.; WOOD, D. E. Branch-and-Bound Methods: A Survey. **Operational Research**, v. 14, p. 699–719, 1966. Available from: https://api.semanticscholar.org/CorpusID:36099120.

LI, M.; VITÁNYI, P. M. Kolmogorov Complexity and its Applications. *In:* ALGORITHMS and Complexity. [*S. l.*]: Elsevier, 1990. p. 187–254. ISBN 978-0-444-88071-0. DOI: 10.1016/B978-0-444-88071-0.50009-6. Available from: https://linkinghub.elsevier.com/retrieve/pii/B9780444880710500096. Visited on: 9 Feb. 2024.

LOURENÇO, H. R.; MARTIN, O. C.; STÜTZLE, T. Iterated Local Search. *In:* HANDBOOK of Metaheuristics. [*S. l.*]: Springer, Boston, MA, 2003. p. 320–353. ISBN 978-0-306-48056-0. DOI: 10.1007/0-306-48056-5_11. Available from:

https://link.springer.com/chapter/10.1007/0-306-48056-5_11. Visited on: 10 Jan. 2024.

LUTZ, R. Evolving good hierarchical decompositions of complex systems. **Journal of System Architecture**, v. 47, p. 613–634, 2001. Available from: https://api.semanticscholar.org/CorpusID:28630106.

MAHDAVI, K.; HARMAN, M.; HIERONS, R. A multiple hill climbing approach to software module clustering. *In:* INTERNATIONAL Conference on Software Maintenance, 2003. ICSM 2003. Proceedings. [*S. l.: s. n.*], Sept. 2003. p. 315–324. ISSN: 1063-6773. DOI: 10.1109/ICSM.2003.1235437. Available from: https://ieeexplore.ieee.org/document/1235437. Visited on: 18 Feb. 2024.

MANCORIDIS, S.; MITCHELL, B.; CHEN, Y.; GANSNER, E. Bunch: A clustering tool for the recovery and maintenance of software system structures. *In:* PROCEEDINGS IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360). [*S. l.*]: IEEE, 1999. p. 50–59.

MANCORIDIS, S.; MITCHELL, B.; RORRES, C.; CHEN, Y.-F.; GANSNER, E. Using Automatic Clustering to Produce High-Level System Organizations of Source Code. *In:* 6TH International Workshop on Program Comprehension (IWPC '98), June 24-26, 1998, Ischia, Italy. Ischia, Italy: IEEE Computer Society, 1998. p. 45–52. ISBN 978-0-8186-8560-6. DOI: 10.1109/WPC.1998.693283. Available from: https://doi.org/10.1109/WPC.1998.693283.

MARTIN, R. C. Design principles and design patterns. **Object Mentor**, v. 1, n. 34, p. 597, 2000.

MAZLAMI, G.; CITO, J.; LEITNER, P. Extraction of Microservices from Monolithic Software Architectures. *In:* PROCEEDINGS of the IEEE International Conference on Web Services (ICWS). [*S. l.: s. n.*], 2017. p. 524–531. DOI: 10.1109/ICWS.2017.61.

MCCONNELL, S. **Code Complete, Second Edition**. [*S. l.: s. n.*], 2004. Available from: https://api.semanticscholar.org/CorpusID:60449735.

MINKU, L. L.; YAO, X. Ensembles and locality: Insight on improving software effort estimation. **Information and Software Technology**, v. 55, p. 1512–1528, 2013. Available from: https://api.semanticscholar.org/CorpusID:584455.

MITCHELL, B. **A Heuristic Search Approach to Solving the Software Clustering Problem**. Mar. 2002. PhD thesis – Drexel University. Visited on: 11 Feb. 2024.

MITCHELL, B.; MANCORIDIS, S. Comparing the decompositions produced by software clustering algorithms using similarity measurements. *In:* PROCEEDINGS IEEE International Conference on Software Maintenance. ICSM 2001. Florence, Italy: IEEE Comput. Soc, 2001. p. 744–753. ISBN 978-0-7695-1189-4. DOI: 10.1109/ICSM.2001.972795. Available from: http://ieeexplore.ieee.org/document/972795/. Visited on: 5 Apr. 2022.

MITCHELL, B.; MANCORIDIS, S. On the Automatic Modularization of Software Systems Using the Bunch Tool. **IEEE Transactions on Software Engineering**, v. 32, n. 3, p. 193–208, Mar. 2006. ISSN 0098-5589. DOI: 10.1109/TSE.2006.31. Available from: https://doi.org/10.1109/TSE.2006.31. Visited on: 12 Feb. 2024.

MITCHELL, B.; MANCORIDIS, S. On the evaluation of the Bunch search-based software modularization algorithm. **Soft Computing**, v. 12, n. 1, p. 77–93, 2008. ISSN 1433-7479. DOI: 10.1007/S00500-007-0218-3. Available from: https://doi.org/10.1007/s00500-007-0218-3.

MITCHELL, B.; MANCORIDIS, S. Using heuristic search techniques to extract design abstractions from source code. *In:* PROCEEDINGS of the 4th Annual Conference on Genetic and Evolutionary Computation. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., July 2002. (GECCO'02), p. 1375–1382. ISBN 978-1-55860-878-8. Visited on: 11 Feb. 2024.

MONÇORES, M. C.; ALVIM, A. C. F.; BARROS, M. O. Large Neighborhood Search applied to the Software Module Clustering problem. **Computers & Operations Research**, v. 91, p. 92–111, Mar. 2018. ISSN 0305-0548. DOI: 10.1016/j.cor.2017.10.004. Available from: https://www.sciencedirect.com/science/article/pii/S0305054817302629. Visited on: 18 Feb. 2024.

MONÇORES, M. d. C. **Busca em vizinhança grande aplicada ao problema de clusterização de módulos de software**. Sept. 2015. MA thesis. Accepted: 2018-06-25T22:06:23Z. Available from: http://www.repositorio-bc.unirio.br:8080/xmlui/handle/unirio/11798. Visited on: 26 Dec. 2023.

PINTO, A. F. **Uma heurística baseada em busca local iterada para o problema de clusterização de módulos de software**. June 2014. MA thesis. Accepted: 2018-07-10T22:00:52Z. Available from: http://www.repositorio-bc.unirio.br:8080/xmlui/handle/unirio/11920. Visited on: 26 Dec. 2023.

POSHYVANYK, D.; MARCUS, A. The Conceptual Coupling Metrics for Object-Oriented Systems. *In:* PROCEEDINGS of the 22nd IEEE International Conference on Software Maintenance (ICSM). [*S. l.: s. n.*], 2006. p. 469–478. DOI: 10.1109/ICSM.2006.67.

PRADITWONG, K. Solving software module clustering problem by evolutionary algorithms. *In:* PROCEEDINGS of the 8th International Joint Conference on Computer Science and Software Engineering (JCSSE). [*S. l.*]: IEEE, 2011. p. 154–159. DOI: 10.1109/JCSSE.2011.5930112.

PRADITWONG, K.; HARMAN, M.; YAO, X. Software Module Clustering as a Multi-Objective Search Problem. **IEEE Transactions on Software Engineering**, v. 37, n. 2, p. 264–282, 2011. DOI: 10.1109/TSE.2010.26.

RISSANEN, J. Modeling by shortest data description. en. **Automatica**, v. 14, n. 5, p. 465–471, Sept. 1978. ISSN 00051098. DOI: 10.1016/0005-1098(78)90005-5. Available from: https://linkinghub.elsevier.com/retrieve/pii/0005109878900055. Visited on: 10 Feb. 2024.

RODRIGUES, R. J. **Uma Abordagem baseada em Incorporação de Palavras para a Remodularização de Software Desenvolvido em Java**. 2025. Tese de Doutorado – Programa de Pós-Graduação em Informática (PPGI), Universidade Federal do Estado do Rio de Janeiro (UNIRIO), Rio de Janeiro, Brasil.

RUKMONO, S. A.; OCHOA, L.; CHAUDRON, M. Deductive Software Architecture Recovery via Chain-of-thought Prompting. *In:* PROCEEDINGS of the ACM/IEEE International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER). [*S. l.: s. n.*], 2024. p. 92–96. DOI: 10.1145/3639476.3639776.

S. SEMAAN, G.; L. V. BOTELHO, A.; S. OCHI, L. HEURÍSTICA BASEADA EM BUSCA LOCAL ITERADA PARA A RESOLUÇÃO DO PROBLEMA DE AGRUPAMENTO DE SISTEMAS ORIENTADOS A OBJETOS. *In:* SIMPÓSIO Brasileiro de Pesquisa Operacional. [*S. l.: s. n.*], 2011. p. 1735–1744.

SALLOU, J.; DURIEUX, T.; PANICHELLA, A. Breaking the Silence: the Threats of Using LLMs in Software Engineering. *In:* PROCEEDINGS of the ACM/IEEE International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER). [*S. l.: s. n.*], 2024. p. 102–106. DOI: 10.1145/3639476.3639764.

SEMAAN, G. S.; OCHI, L. S. Algoritmo Evolutivo para o Problema de Clusterização em Grafos Orientados. **Simpósio de Pesquisa Operacional e Logística da Marinha (SPOLM2007). Rio de Janeiro, Brasil:[sn]**, 2007.

SHANNON, C. E. A Mathematical Theory of Communication. en. **The Bell System Technical Journal**, v. 27, p. 623–656, Oct. 1948.

SHTERN, M.; TZERPOS, V. A Framework for the Comparison of Nested Software Decompositions. *In:* 11TH Working Conference on Reverse Engineering, WCRE 2004, Delft, The Netherlands, November 8-12, 2004. [*S. l.*]: IEEE Computer Society, 2004. p. 284–292. DOI: 10 . 1109 / WCRE . 2004 . 6. Available from: https://doi.org/10.1109/WCRE.2004.6.

SILVA, L. L.; VALENTE, M. T.; MAIA, M. d. A. Co-change Clusters: Extraction and Application on Assessing Software Modularity. *In:* TRANSACTIONS on Aspect-Oriented Software Development XII. [*S. l.*]: Springer, 2015. (Lecture Notes in Computer Science). p. 96–131. DOI: 10.1007/978-3-662-46734-3_3.

SIMONS, C. L.; PARMEE, I. C. Elegant object-oriented software design via interactive, evolutionary computation. **IEEE Transactions on Systems, Man, and**

Cybernetics, Part C (Applications and Reviews), v. 42, n. 6, p. 1797–1805, 2012. Publisher: IEEE.

TAN, P.-N.; KUMAR, V.; SRIVASTAVA, J. Selecting the right objective measure for association analysis. **Information Systems**, v. 29, n. 4, p. 293–313, 2004. DOI: 10.1016/S0306-4379(03)00072-3.

TUCKER, A.; SWIFT, S.; LIU, X. Variable grouping in multivariate time series via correlation. **IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)**, v. 31, n. 2, p. 235–245, Apr. 2001. Conference Name: IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics). ISSN 1941-0492. DOI: 10.1109/3477.915346. Available from: https://ieeexplore.ieee.org/document/915346/similar#similar. Visited on: 18 Feb. 2024.

TZERPOS, V.; HOLT, R. ACCD: an algorithm for comprehension-driven clustering. *In:* PROCEEDINGS Seventh Working Conference on Reverse Engineering. [*S. l.: s. n.*], Nov. 2000. p. 258–267. ISSN: 1095-1350. DOI: 10.1109/WCRE.2000.891477. Available from: https://ieeexplore.ieee.org/document/891477. Visited on: 18 Feb. 2024.

TZERPOS, V.; HOLT, R. MoJo: a distance metric for software clusterings. *In:* SIXTH Working Conference on Reverse Engineering (Cat. No.PR00303). [*S. l.: s. n.*], 1999. p. 187–193. DOI: 10.1109/WCRE.1999.806959.

WEN, Z.; TZERPOS, V. An Effectiveness Measure for Software Clustering Algorithms. *In:* 12TH International Workshop on Program Comprehension (IWPC 2004), 24-26 June 2004, Bari, Italy. [*S. l.*]: IEEE Computer Society, 2004. p. 194–203. DOI: 10.1109/WPC.2004.1311061. Available from: https://doi.org/10.1109/WPC.2004.1311061.

WEN, Z.; TZERPOS, V. An Optimal Algorithm for MoJo Distance. *In:* 11TH International Workshop on Program Comprehension (IWPC 2003), May 10-11, 2003, Portland, Oregon, USA. [*S. l.*]: IEEE Computer Society, 2003. p. 227–236. DOI: 10.1109/WPC.2003.1199206. Available from: https://doi.org/10.1109/WPC.2003.1199206.

WEYUKER, E. Evaluating software complexity measures. **IEEE Transactions on Software Engineering**, v. 14, n. 9, p. 1357–1365, Sept. 1988. Conference Name: IEEE Transactions on Software Engineering. ISSN 1939-3520. DOI: 10.1109/32.6178. Available from: https://ieeexplore.ieee.org/document/6178?denied=. Visited on: 3 Mar. 2024.

WOHLIN, C. **Experimentation in software engineering**. New York: Springer, 2012. ISBN 978-3-642-29043-5.

WOOD, J. A. **Improving software designs via the minimum description length principle**. 1998. PhD Thesis. Available from: https://api.semanticscholar.org/CorpusID:38339071.

XUAN, J.; JIANG, H.; REN, Z.; LUO, Z. Solving the Large Scale Next Release Problem with a Backbone-Based Multilevel Algorithm. **IEEE Transactions on Software Engineering**, v. 38, n. 5, p. 1195–1212, 2012. DOI: 10.1109/TSE.2011.92.

YING, A. T. T.; MURPHY, G. C.; NG, R.; CHU-CARROLL, M. C. Predicting source code changes by mining change history. **IEEE Transactions on Software Engineering**, v. 30, n. 9, p. 574–586, 2004. DOI: 10.1109/TSE.2004.52.

YOURDON, E.; CONSTANTINE, L. L. Structured design. Fundamentals of a discipline of computer program and systems design. en. **Englewood Cliffs: Yourdon Press**, 1979. Available from: https://ui.adsabs.harvard.edu/abs/1979sdfd.book.....Y/abstract. Visited on: 30 Jan. 2024.

ZHANG, K.; SHASHA, D. Simple Fast Algorithms for the Editing Distance between Trees and Related Problems. **SIAM Journal on Computing**, v. 18, n. 6, p. 1245–1262, 1989. _eprint: https://doi.org/10.1137/0218082. DOI: 10.1137/0218082. Available from: https://doi.org/10.1137/0218082.

ZHANG, Y.; LIU, C.; SUN, Y.; XU, Z.; SUN, W.; LI, W.; JIN, W.; LIU, Y. Semantic-Enhanced Automatic Refinement of Architecture Recovery Results Using LLMs. *In:* PROCEEDINGS of the 48th IEEE/ACM International Conference on Software Engineering (ICSE). [*S. l.: s. n.*], 2026. p. 1687–1699. DOI: 10.1145/3744916.3773114.

ZIMMERMANN, T.; DIEHL, S.; ZELLER, A. How history justifies system architecture (or not). *In:* PROCEEDINGS of the 6th International Workshop on Principles of Software Evolution (IWPSE). [*S. l.: s. n.*], 2003. p. 73–83. DOI: 10.1109/IWPSE.2003.1231213.

ZIMMERMANN, T.; ZELLER, A.; WEISSGERBER, P.; DIEHL, S. Mining version histories to guide software changes. **IEEE Transactions on Software Engineering**, v. 31, n. 6, p. 429–445, 2005. DOI: 10.1109/TSE.2005.72.

APPENDIX A LIST OF EXAMPLE COMMITS OF A HYPOTHETICAL SYSTEM

```
commit #01
Author: Leo Antunes <leo.antunes@edu.unirio.br>
Date: Mon Mar 17 22:02:39 2025 +0000
Comment: initial commit

A    c0/c1/c1-1/m1.java
A    c0/c1/m2.java
A    c0/c1/c1-1/m3.java
A    c0/c1/m4.java
A    c0/c2/c2-1/m5.java
A    c0/c2/c2-1/m6.java
A    c0/c2/m7.java
A    c0/c2/c2-1/m8.java
```

Listing A.1: commit #01

```
commit #02
Author: Leo Antunes <leo.antunes@edu.unirio.br>
Date: Mon Mar 18 19:47:25 2025 +0000
Comment: update #01

M       c0/c1/c1-1/m3.java
M       c0/c1/m4.java
```

Listing A.2: commit #02

```
commit #03
Author: Leo Antunes <leo.antunes@edu.unirio.br>
Date: Mon Mar 19 18:36:54 2025 +0000
Comment: update #02

M       c0/c2/c2-1/m6.java
M       c0/c2/c2-1/m8.java
```

Listing A.3: commit #03

```
commit #04
Author: Leo Antunes <leo.antunes@edu.unirio.br>
Date: Mon Mar 20 20:03:45 2025 +0000
```

```
Comment: update #03

M       c0/c1/c1-1/m1.java
M       c0/c2/c2-1/m5.java
M       c0/c2/m7.java
```

Listing A.4: commit #04

```
commit #05
Author: Leo Antunes <leo.antunes@edu.unirio.br>
Date: Mon Mar 21 17:32:23 2025 +0000
Comment: update #04

M       c0/c1/c1-1/m1.java
M       c0/c1/c1-1/m3.java
M       c0/c1/m4.java
```

Listing A.5: commit #05

```
commit #06
Author: Leo Antunes <leo.antunes@edu.unirio.br>
Date: Mon Mar 22 16:29:32 2025 +0000
Comment: update #05

M       c0/c1/c1-1/m3.java
M       c0/c2/c2-1/m5.java
M       c0/c2/m7.java
```

Listing A.6: commit #06

```
commit #07
Author: Leo Antunes <leo.antunes@edu.unirio.br>
Date: Mon Mar 23 19:07:51 2025 +0000
Comment: update #06

M       c0/c1/c1-1/m1.java
M       c0/c1/c1-1/m3.java
M       c0/c2/m7.java
```

Listing A.7: commit #07

```
commit #08
Author: Leo Antunes <leo.antunes@edu.unirio.br>
Date: Mon Mar 24 22:42:15 2025 +0000
Comment: update #07
```

```
5
6 M       c0/c1/c1-1/m1.java
7 M       c0/c1/c1-1/m3.java
8 M       c0/c1/m4.java
```

Listing A.8: commit #08

```
1 commit #09
2 Author: Leo Antunes <leo.antunes@edu.unirio.br>
3 Date: Mon Mar 25 21:15:13 2025 +0000
4 Comment: update #08
5
6 M       c0/c1/m2.java
7 M       c0/c1/c1-1/m3.java
8 M       c0/c1/m4.java
```

Listing A.9: commit #09

```
1 commit #10
2 Author: Leo Antunes <leo.antunes@edu.unirio.br>
3 Date: Mon Mar 26 15:08:31 2025 +0000
4 Comment: update #09
5
6 M       c0/c2/c2-1/m5.java
7 M       c0/c2/c2-1/m6.java
```

Listing A.10: commit #10

```
1 commit #11
2 Author: Leo Antunes <leo.antunes@edu.unirio.br>
3 Date: Mon Mar 27 18:46:22 2025 +0000
4 Comment: update #10
5
6 M       c0/c1/m2.java
7 M       c0/c2/m7.java
```

Listing A.11: commit #11

```
1 commit #12
2 Author: Leo Antunes <leo.antunes@edu.unirio.br>
3 Date: Mon Mar 28 23:51:55 2025 +0000
4 Comment: update #11
5
6 M       c0/c1/c1-1/m1.java
7 M       c0/c2/c2-1/m6.java
```

```
M       c0/c2/c2-1/m8.java
```

Listing A.12: commit #12

```
commit #13
Author: Leo Antunes <leo.antunes@edu.unirio.br>
Date: Mon Mar 29 16:38:43 2025 +0000
Comment: update #12

M       c0/c2/c2-1/m5.java
M       c0/c2/c2-1/m6.java
M       c0/c2/c2-1/m8.java
```

Listing A.13: commit #13

```
commit #14
Author: Leo Antunes <leo.antunes@edu.unirio.br>
Date: Mon Mar 30 15:43:27 2025 +0000
Comment: update #13

M       c0/c1/c1-1/m1.java
M       c0/c1/m2.java
M       c0/c2/m7.java
```

Listing A.14: commit #14

```
commit #15
Author: Leo Antunes <leo.antunes@edu.unirio.br>
Date: Mon Mar 31 13:35:49 2025 +0000
Comment: update #14

M       c0/c1/c1-1/m1.java
M       c0/c1/m2.java
```

Listing A.15: commit #15

```
commit #16
Author: Leo Antunes <leo.antunes@edu.unirio.br>
Date: Mon Apr 01 06:27:59 2025 +0000
Comment: update #15

M       c0/c1/m2.java
M       c0/c2/c2-1/m6.java
M       c0/c2/m7.java
```

Listing A.16: commit #16

```
commit #17
Author: Leo Antunes <leo.antunes@edu.unirio.br>
Date: Mon Apr 02 14:06:11 2025 +0000
Comment: update #16

M       c0/c2/c2-1/m5.java
M       c0/c2/m7.java
M       c0/c2/c2-1/m8.java
```

Listing A.17: commit #17

```
commit #18
Author: Leo Antunes <leo.antunes@edu.unirio.br>
Date: Mon Apr 03 19:21:14 2025 +0000
Comment: update #17

M       c0/c1/c1-1/m1.java
M       c0/c1/m2.java
M       c0/c2/c2-1/m5.java
```

Listing A.18: commit #18

```
commit #19
Author: Leo Antunes <leo.antunes@edu.unirio.br>
Date: Mon Apr 04 14:57:17 2025 +0000
Comment: update #18

M       c0/c1/m2.java
M       c0/c1/c1-1/m3.java
M       c0/c1/m4.java
```

Listing A.19: commit #19

```
commit #20
Author: Leo Antunes <leo.antunes@edu.unirio.br>
Date: Mon Apr 05 10:45:54 2025 +0000
Comment: update #19

M       c0/c2/c2-1/m6.java
M       c0/c2/m7.java
M       c0/c2/c2-1/m8.java
```

Listing A.20: commit #20