MODULARIZATION OF SOFTWARE LIBRARIES

BURAK ÖYKE

ÖZYEĞİN UNIVERSITY

JUNE, 2025

MODULARIZATION OF SOFTWARE LIBRARIES

by
Burak Öyke

Thesis
Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science

in
Computer Science

Advisor: Prof. Hasan Sözer

Graduate School of Science and Engineering
Özyeğin University
İstanbul

June, 2025

MODULARIZATION OF SOFTWARE LIBRARIES

Approved by:

Prof. Hasan Sözer, Advisor
Department of Computer Science
Özyeğin University

Prof. Okan Örsan Özener
Department of Industrial Engineering
Özyeğin University

Prof. Geylani Kardaş
International Computer Institute
Ege University

Approval Date: May 13, 2025

DECLARATION OF ORIGINALITY

I hereby declare that I am the sole author of this thesis and that this is the true copy of my thesis, including the final revisions, approved by my thesis committee. All data and information have been obtained, produced, and presented in accordance with the rules of research ethics and principles of academic honesty. As required by these rules, to the best of my knowledge, I have acknowledged ideas, thoughts, and any copyrighted material in accordance with the standard referencing rules. I certify that any part of this thesis has not been submitted for a degree or diploma in another educational institution.

Burak ÖYKE

ABSTRACT

Most existing software modularization approaches focus on balancing cohesion and coupling to cluster software modules effectively. However, the criteria for modularizing software libraries should be addressed differently, given their unique characteristics. In these systems, cohesion emerges as the primary criterion, defined by the common usage patterns of modules rather than their interdependencies. Modules are grouped not because of strong coupling with each other but because they are often used together by other modules. We propose a novel optimization model based on this perspective and evaluate its effectiveness with two case studies. The first study involves an industrial application from the consumer electronics domain. The second study examines the transition from Java 8 to the Java 9 Platform Module System, where existing packages are reorganized into a higher-level unit of packaging and encapsulation, called modules. Our evaluation focuses on the efficiency of clustering, measured by the number of clusters required by modules and the reduction in redundant module imports. By grouping modules into clusters, importing a few clusters makes all its modules available, reducing overall imports. However, the goal is to balance this reduction by ensuring that only relevant modules are included in each cluster to prevent redundancy. We compare the results of our model against manually created clusters and alternative solutions generated by existing clustering algorithms, demonstrating significant performance improvements.

Keywords: software module clustering, software modularization, integer programming, software libraries, industrial case study

ÖZET

Yazılım kümeleme yaklaşımlarının çoğu, etkili bir kümeleme için uyumluluk ve bağımlılığı dengelemeye odaklanmaktadır. Ancak, yazılım kütüphanelerinin benzersiz özellikleri göz önünde bulundurulduğunda kümeleme ölçütleri farklı şekilde ele alınmalıdır. Bu sistemlerde, modüllerin birbirine olan bağımlılıkları yerine ortak kullanım kalıpları tarafından tanımlanan uyumluluk birincil kriter olarak ortaya çıkmaktadır. Modüller, birbirleriyle güçlü bir şekilde uyumluluğu için değil, genellikle diğer modüller tarafından birlikte kullanıldıkları için gruplandırılmıştır. Bu bakış açısına göre yeni bir optimizasyon modeli önerilmiştir ve etkinliğini iki vaka çalışmasıyla değerlendirilmiştir. İlk çalışma, tüketici elektroniği alanından bir endüstriyel uygulamayı içermektedir. İkinci çalışma, mevcut paketlerin modüller adı verilen daha üst düzey bir paketleme ve kapsülleme birimine yeniden düzenlendiği Java 8'den Java 9 Platform Modül Sistemine geçişi incelemektedir. Değerlendirme, modüller tarafından ihtiyaç duyulan küme sayısı ve yinelenen modül içe aktarmalarının azaltılmasıyla ölçülen kümeleme verimliliğine odaklanmaktadır. Modüller kümelere ayrılarak gruplanınca, birkaç kümeyi içe aktarmak tüm modüllerini erişilebilir hale getirir ve genel içe aktarım sayısnı azaltmaktadır. Ancak bu azalmayı dengelemek amacıyla, her kümeye yalnızca o kümeyle ilgili modüller dahil edilmiştir. Modelimizin sonuçlarını, elle oluşturulan kümelerle ve mevcut kümeleme algoritmaları tarafından üretilen alternatif çözümlerle karşılaştırarak önemli performans iyileştirmeleri gösterilmiştir.

Anahtar Kelimeler: yazılım kümeleme, modüler yazılım tasarımı, tamsayı programlama, yazılım kütüphaneleri, endüstriyel vaka çalışması

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my advisor, Prof. Hasan Sözer, for his invaluable guidance and support throughout this journey. I am also grateful to Vestel Electronics for sharing their codebase with me, as well as for supporting my experiments.

I would also like to express my sincere gratitude to Prof. Okan Örsan Özener from the Department of Industrial Engineering at Özyeğin University, who served as the second reader of this thesis, and Prof. Geylani Kardaş from the Department of Computer Science at Ege University, who served as the third reader. I am deeply indebted to them for their invaluable insights and constructive feedback, which have greatly contributed to the quality of this work.

Finally, I could not have completed this thesis without the unwavering support of my beloved wife, Merve. Her immense understanding, patience, and encouragement over the past few years have been instrumental in this journey. I am forever grateful for her love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS AND ABBREVIATIONS

JDK	Java Development Kit
JAR	Java ARchive
JPMS	Java Platform Module System
ILP	Integer Linear Programming
MM1	Mathematical Model 1
MM2	Mathematical Model 2
Java SE	Java Standard Edition
HAL	Hardware Abstraction Layer
JNI	Java Native Interface
API	Application Programming Iterface
RSF	Rigi Standard Format
DSM	Dependency Structure Matrix
NAHC	Next Ascent Hill Climbing
SAHC	Steepest Ascent Hill Climbing
LIMBO	scaLable InforMation BOttleneck
ACDC	Algorithm for Comprehension-Driven Clustering (4
MGMC	Multi-level Greedy Modularity Clustering
MoJoFM	Move or Join eFfectiveness Measure
a2a	Architecture to Architecture
NCI	Number of Clusters Imported

NUFI Number of Unnecessary Files Imported

1. INTRODUCTION

Software module clustering is employed as an automated approach to cluster software entities such as classes, modules, or files [2]. The goal of clustering is to simplify the complexity of a large software system by grouping multiple artifacts into a single cluster, which serves as an abstract representation of the grouped artifacts [3]. This resulting decomposition enhances the comprehensibility of the system [4–6], helps in the identification of design anomalies [7, 8], and provides insights for refactoring opportunities [9, 10].

Figure 1.1: *A conceptual overview of the software module clustering process.*

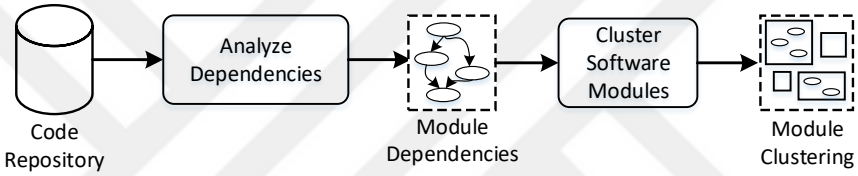
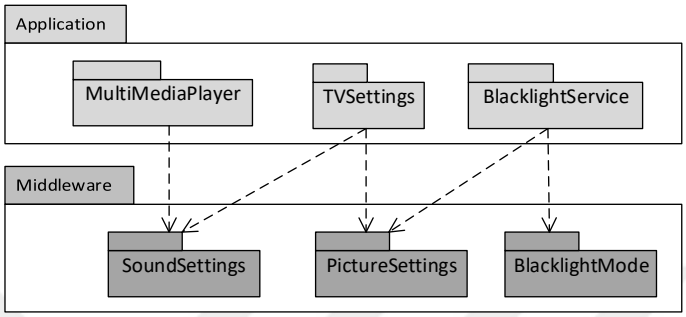


Figure 1.1 illustrates the software clustering process [11]. Hereby, the goal of clustering is mainly to maximize modularity [12], which is achieved by minimizing coupling and maximizing cohesion. Coupling is measured as the number of dependencies among artifacts that are placed in different clusters. Cohesion is measured as the number of dependencies among artifacts that are placed in the same clusters. Existing software modularization approaches mainly aim at balancing cohesion and coupling to cluster software modules. However, we argue that the clustering criteria for modularizing software libraries should be reconsidered. In these systems, cohesion emerges as the primary criterion, defined by the common usage patterns of modules rather than their interdependencies. Modules are grouped not because of strong coupling with each other but because they are frequently used together by other modules. Figure 1.2 depicts an illustrative example, where a set of packages (e.g., *SoundSettings*, *PictureSettings*, *BlacklightMode*) are grouped together in a higher level package (e.g., *Middleware*) although they do not

directly depend on each other. Such indirect relationships can play a more decisive role in shaping the architectural organization of some software systems [5, 13, 14].

Figure 1.2: *An illustrative example of common package dependencies among modules in a representative portion of an Android TV software stack, highlighting groups of modules that are clustered together although they do not directly depend on each other.*



It was recently concluded that experts do not necessarily cluster programs based on cohesion and coupling only [15]. Another evidence that supports our argument comes from the evolution of the Java Development Kit (JDK). JDK is originally organized in packages that can be compiled in Java ARchive (JAR) files. These packages can be imported in Java applications by specifying the corresponding *classpath*. A Java classpath may contain several libraries each defining a class with the same fully qualified name. At runtime, the class loading mechanism resolves these redundancies by allowing only the first encountered class version to be loaded per class loader. This makes it difficult to foresee which class will actually be used, resulting in unpredictable program behaviour. This problem is sometime called "JAR hell" or "classpath hell" [16]. To mitigate this problem and to manage an increasing number of packages, Java Platform Module System (JPMS) was introduced, while transitioning from Java version 8 to version 9. JPMS reorganizes packages into a set of clusters so-called *modules*, where each Java module serves as a higher-level unit of packaging and encapsulation (See Figure 4.1). One of the largest modules in JPMS, *java.base* has 160 packages and 32 of these do not depend on any other package in the *java.base* module.

Software libraries are generally organized and clustered based on common usage patterns. As a library grows, the number of package imports can increase to hundreds, making it challenging to manage and track the imported packages. Grouping related packages into higher-level encapsulation units, such as Java modules, provides a practical solution. This approach allows a cluster of related packages to be imported as a single unit, rather than importing each package individually. However, importing unnecessary packages bundled within a cluster can be undesirable. This is particularly problematic in embedded systems, where resources are limited, and packages are loaded at runtime. Importing redundant packages can also disrupt incremental builds, leading to inefficiencies. We propose a novel optimization approach based on this perspective.

We evaluate the effectiveness of our model through two case studies. The first study involves an industrial application from the consumer electronics domain. The second study examines the transition from Java 8 to the JPMS in Java 9. We use two metrics to evaluate the effectiveness of the clustering performed. The first metric assesses the total number of clusters that need to be imported as a result of the clustering. The second metric measures the number of unnecessary files imported along with the required clusters. These metrics help quantify the balance between reducing the complexity of imports and minimizing redundancy. We also compare the results of our model against manually created clusters and alternative solutions generated by existing clustering algorithms employed in the software architecture recovery and software modularization domain. The results show that our approach reduces both the total number of imports and the number of unnecessary packages imported.

The remainder of this paper is structured as follows: Related work is summarized in the next section. In Section 3, we introduce our optimization model. The experimental setup used for the evaluation of this model is detailed in Section 4. Results are presented and discussed in Section 5. Finally, we conclude the paper in Section 6.

2. RELATED WORK

There have been various approaches introduced for software architecture recovery [17,18] and software module clustering [2] in the last couple of decades. The modularity clustering problem is known to be NP-hard [19]. Therefore, many heuristic and meta-heuristic algorithms are proposed. Hierarchical [20] and greedy algorithms [11, 21–23] merge clusters with similar modules in successive phases based on a similarity metric. On the other hand, meta-heuristic and search-based algorithms treat software module clustering as a combinatorial optimization problem. Genetic algorithms are commonly employed in this category of algorithms [24–28]. In this paper, we propose an integer programming model for clustering. Although the problem is NP-hard, the model can provide solutions for clustering software libraries, where clustering is performed at a higher level of abstraction (i.e., packages) than source code (i.e., functions or classes), effectively handling hundreds of artifacts.

Software module clustering algorithms take artifact dependency graphs (See Figure 1.1) as input, which can also be represented as design structure matrices [29]. In this work, we utilize *include dependencies* [6] as input. These dependencies are established if one file/package declares that it includes/imports another file/package [6]. We ignore low level code dependencies such as symbol dependencies and function calls [6]. There are also semantic-based clustering algorithms [30] and various other types of dependencies considered for clustering [31]. These dependencies are obtained from the analysis of various artifacts including documentation [32, 33], databases [5, 13] and code repositories [34, 35].

The main objective adopted by clustering algorithms is usually to maximize modularity [36] although multiple objectives and constraints can be adopted [37, 38] considering, for instance, the number and size of the clusters. Existing software modularization approaches mainly aim at balancing cohesion and coupling to maximize modularity.

However, the clustering criteria can differ for different types of software systems. The way that cohesion and coupling are measured can also be different. The need for new/additional criteria for software modularization was recently pointed out by Pourasghar et al. [15]. They introduce two new criteria based on the program control flow. These criteria aim at collapsing the set of input and output dependencies of each cluster within a single or a small number of nodes, which is aligned with the basic principle adopted for the Facade design pattern [39]. In this work, we focus on modularizing software libraries. In these systems, modules are grouped based on common usage patterns rather than their interdependencies. In the following section, we introduce an optimization model for clustering software modules from this perspective.

3. THE OPTIMIZATION MODEL

We formulate the clustering problem as an Integer Linear Programming (ILP) model. The solution to this formulation yields an optimal assignment of packages to clusters under a set of structural and dependency constraints. However, the problem is inherently complex due to its "multi-objective nature". Specifically, the model addresses two conflicting objectives: *(i)* minimizing the total number of clusters that are imported and *(ii)* minimizing the number of unnecessary files (packages) imported as a result of cluster assignments. Balancing these objectives is non-trivial, as improving one may negatively impact the other.

In multi-criteria decision-making problems of this type, two widely recognized modeling strategies can be adopted [40]. The first is the "weighted sum approach", in which a single objective function is constructed by assigning a weight to each objective and combining them into one aggregate objective function. The primary advantage of this approach is its simplicity and flexibility; it allows the decision-maker to express the relative importance of each criterion directly through the weights. Furthermore, it enables the use of standard optimization solvers without the need for structural changes to the model. However, this method has notable drawbacks. Selecting appropriate weights can be challenging and subjective, often requiring extensive sensitivity analysis. Moreover, the weighted sum approach may fail to capture the true trade-offs between objectives, especially when the objectives are measured in different units or scales. It may also struggle in scenarios where the Pareto front is non-convex, leading to potentially suboptimal or biased solutions [41].

The second strategy is the "hierarchical approach", which prioritizes the objectives by optimizing them sequentially according to their importance. In this method, the most critical objective is optimized first. Once its optimal value is obtained, the second objective is optimized in the feasible region defined by the optimal solution of the

primary objective, and so on. The hierarchical approach has the advantage of eliminating the need to define subjective weights and allows the decision-maker to strictly enforce priority among objectives. It guarantees that the primary objective is not compromised for the sake of secondary criteria. Nevertheless, this method can limit flexibility, as lower-priority objectives are only optimized within the solution space determined by the higher-priority objective. It may also increase computational effort due to the need for multiple optimization runs or iterative procedures.

In this study, we adopt a modified version of the hierarchical optimization approach. Specifically, we first prioritize the minimization of the total number of cluster imports, which serves as the primary objective. Subsequently, instead of directly incorporating the secondary objective into the model, we experiment with varying upper bound values for the total number of package imports as explained in the following and later demonstrated in Section 5.3. This allows us to systematically explore the trade-off between the two objectives and approximate the Pareto frontier. The upper bound values are determined iteratively, starting from a benchmark obtained by solving an alternative formulation of the model in which the objective is to minimize the total number of files (packages) imported, regardless of the number of cluster imports. The optimal solution value from this alternative model serves as a lower bound reference point, and the upper bound is gradually increased to investigate the feasible trade-offs.

This approach ensures that the structural efficiency of the clustering scheme quantified by the total number of cluster imports—is preserved as the primary optimization goal, while still allowing control over the number of package imports. It also provides valuable insights into the inherent trade-offs between reducing cluster imports and limiting package imports, which is crucial for practical decision-making.

It is worth noting that, throughout this study, we use the terms “minimizing the

number of total packages imported" and "minimizing the number of unnecessary packages imported" interchangeably. This is because the difference between these two quantities is constant and corresponds to the minimum number of required packages imposed by the package dependency.

Algorithm 1 Multi-criteria Optimization Approach

1: Solve MM1 to obtain z^*_{MM1}, the minimum number of total packages imported.
2: Create an Upper Bound Value Set $\mathbb{U} = \{z^*_{MM1}, z^*_{MM1} + \alpha, z^*_{MM1} + 2\alpha, \dots\}$ where α is a constant scalar.
3: **for** $U \in \mathbb{U}$ **do**
4: Solve $MM2(U)$ to obtain $z^*_{MM2}(U)$, the minimum total number of clusters imported.
5: **end for**
6: Obtain a set of solutions to construct the Pareto frontier.

Our overall approach is outlined in Algorithm 1. We define two optimization models, Mathematical Model 1 (MM1) and Mathematical Model 2 (MM2), as described below. By grouping packages into disjoint clusters, we aim to optimize two criteria. The first criterion is to acquire the required packages by importing the minimum number of clusters, which is addressed by MM2. The second criterion is to minimize the total number of packages included in the imported clusters, which is handled by MM1. Initially, we solve MM1 to obtain the minimum number of packages that need to be imported. This value serves as an upper bound, U, for the number of packages to be imported and this limit is incorporated as a constraint in MM2. To explore a broader search space, we define a set of values, $\mathbb{U}$, by incrementally adding multiples of a scalar α to the minimum value. This allows for a stepwise relaxation of the constraint. For each value in $\mathbb{U}$, we solve MM2, and the resulting solutions are used for defining the Pareto Frontier. The size of $\mathbb{U}$ defines the number of iterations.

In our evaluation, we report the results for a single iteration of our approach for comparison with manual clustering and alternative algorithms. That is, a single U value as obtained with MM1 is used for solving MM2, without considering any relaxation of

the upper bound. The corresponding results are discussed in Section 5.1 and Section 5.2. Then, we apply the approach with $|\mathbb{U}| = 7$ and α as approximately 10% of the number of packages as discussed in Section 5.3 to demonstrate the versatility of the approach.

3.1 Mathematical Model 1

3.1.1 Sets and Indices

- $i \in \{1, 2, \ldots, n_{\text{Packages}}\}$: Index of packages
- $j \in \{1, 2, \ldots, n_{\text{Packages}}\}$: Index of packages (for dependency)
- $k \in \{1, 2, \ldots, n_{\text{Clusters}}\}$: Index of clusters

3.1.2 Parameters

- PackageDependency$[i][j]$: Equals 1 if package i depends on package j, 0 otherwise
- M : A large constant

3.1.3 Decision Variables

- $x_{ik} \in \{0, 1\}$: 1 if package i is assigned to cluster k, 0 otherwise
- $y_{ik} \in \{0, 1\}$: 1 if package i calls cluster k, 0 otherwise
- $z_k \in \mathbb{Z}_+$: Number of times cluster k is called
- $P_{ik} \in \mathbb{R}_+$: Number of packages in cluster k imported by package i
- $SP_i \in \mathbb{R}_+$: Total number of packages imported by package i

3.1.4 Objective Function

$$z^*_{MM1} = \min \sum_{i=1}^{n_{\text{Packages}}} \sum_{k=1}^{n_{\text{Clusters}}} P_{ik} \tag{3.1}$$

3.1.5 Constraints

1. Dependency

$$\sum_{j=1}^{n_{\text{Packages}}} x_{jk} \times \text{PackageDependency}[i][j] \leq M \times y_{ik}$$

$$\forall i \in \{1, \ldots, n_{\text{Packages}}\}, \quad k \in \{1, \ldots, n_{\text{Clusters}}\} \quad (3.2)$$

2. Unique Cluster Assignment

$$\sum_{k=1}^{n_{\text{Clusters}}} x_{ik} = 1 \quad \forall i \in \{1, \ldots, n_{\text{Packages}}\} \quad (3.3)$$

3. Cluster Call Count

$$\sum_{i=1}^{n_{\text{Packages}}} y_{ik} - z_k \leq 0 \quad \forall k \in \{1, \ldots, n_{\text{Clusters}}\} \quad (3.4)$$

4. Cluster Package Count

$$\sum_{j=1}^{n_{\text{Packages}}} x_{jk} \leq P_{ik} + M \times (1 - y_{ik}) \quad \forall i \in \{1, \ldots, n_{\text{Packages}}\},\ k \in \{1, \ldots, n_{\text{Clusters}}\} \quad (3.5)$$

5. Symmetry Breaking

$$z_{k+1} - z_k \leq 0 \quad \forall k \in \{1, \ldots, n_{\text{Clusters}} - 1\} \quad (3.6)$$

MM1 aims to optimize the assignment of packages to clusters while minimizing the total number of packages imported as the objective function (Equation 3.1). Constraints (1) ensure that a package imports a cluster if it depends on at least one package assigned to that cluster. Constraints (2) guarantee that each package is assigned to exactly one cluster. Constraints (3) ensure that if any package imports a cluster, the corresponding cluster import count variable is updated accordingly. Constraints (4) determine how many packages imported by a given package based on the clusters formed. The formation of this constraint avoids non-linearity. Finally, a symmetry-breaking constraint is

added to eliminate equivalent solutions that differ only in the ordering of clusters. These constraints force the cluster import counts to follow a non-increasing sequence, which improves computational efficiency by reducing the solution space without affecting the optimality of the solution.

Next, we present the second formulation. For brevity and clarity, we only highlight the changes relative to the previous formulation.

3.2 Mathematical Model 2

3.2.1 Parameters

- U : An upper bound on the number of total files (packages) imported

3.2.2 Objective Function

$$z^*_{MM2}(U) = \min \sum_{k=1}^{n_{\text{Clusters}}} z_k \tag{3.7}$$

3.2.3 Additional Constraints

6. Total Package Call Limit

$$\sum_{i=1}^{n_{\text{Packages}}} \sum_{k=1}^{n_{\text{Clusters}}} P_{ik} \leq U \tag{3.8}$$

MM2 aims to optimize the assignment of packages to clusters, while minimizing the total number of cluster imports. The objective function minimizes the total number of clusters that are imported, thereby encouraging a compact clustering structure with fewer imports. All the 5 constraints defined for MM1 applies to this model as well. An additional 6^{th} constraint (Equation 3.8) calculates the total number of packages imported and limit this summation by the given upper bound. Recall that this upper bound is obtained by solving the other model, MM1. We present the evaluation of our approach in the following section.

4. EVALUATION

We evaluated our model with two different experimental objects. The first one is a library employed by industrial applications from the consumer electronics domain. We constructed a dataset from this library that includes both the original library and a second version of it that is clustered by a domain expert. The second dataset is retrieved from the JDK as part of the open source Java Standard Edition (Java SE) platform dating back to 2017, when a transition was made from the monolithic JDK of Java 8 to JPMS introduced by Java 9. We selected these two libraries because they were already subject to manual modularization performed by domain experts. We compared the manual clustering of libraries with respect to the automatically created clusters. In the following, we first explain the created datasets. Then, we explain the alternative clustering algorithms used for comparison. We conclude the section with the list of metrics used for comparison and evaluation.

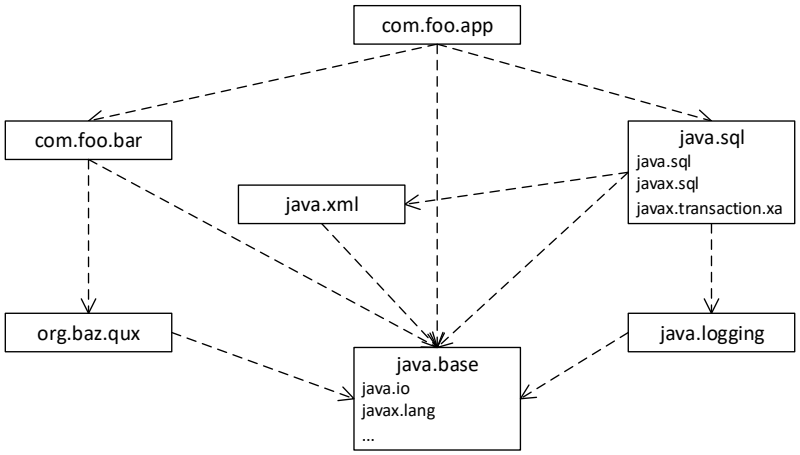
4.1 Experimental Objects and Data Collection

Our first case study focuses on Android TV systems being developed and maintained at Vestel[1], which is one the largest consumer electronics companies in Europe. Vestel produces TV products for 157 different brands from 145 different countries [42]. The Android TV platform employed by these products uses native code for high performance media operations, hardware-controlled services, and management of manufacturer-specific features. Such operations are usually performed through Hardware Abstraction Layer (HAL) and system libraries written in C/C++. Java Native Interface (JNI) is used to access these native components within the Android operating system. JNI bridges the

[1] `http://www.vestel.com.tr`

framework classes in the Java layer with low-level native codes and enables the management of hardware-controlled operations via high-level Application Programming Interface (API)s. In Android TV projects, classes that collect information from the native layer or access system resources with JNI are usually modularized and packaged in the Android Library format. This architectural structure both abstracts developers from the complexity of native code and allows hardware-dependent components to be managed in a modular and sustainable manner. The Java library that we used in this study is provided by the chip vendor and it consists of 137 packages, which are imported by various modules developed by Vestel. These imports are performed by including the Java file of the corresponding package. Therefore, we use the terms 'file' and 'package' interchangeably. The number of files included is equal to the number of packages imported. The list of imports can account for hundreds of lines of code for some modules. Stakeholders in the company decided to cluster these packages to organize them and simplify their usage. Clustering allows for a set of related packages to be imported as a single unit; however, importing unnecessary packages bundled within a cluster is undesirable. Android TV systems are limited in resources and the inclusion of redundant packages impacts both the load time and the built time. An expert within the company came up with a clustering in the year 2024 to balance these concerns. We documented the clustering results in a file with Rigi Standard Format (RSF) format [43,44]. We derived package dependencies by parsing the import statements and store the resulting dependency graph in RSF format as well. The dependency graph represents a Dependency Structure Matrix (DSM) [45], where each cell indicates the presence of a dependency from one entity to another, corresponding to the row and column of the matrix, respectively. The DSMs stored in RSF files are used as input to the clustering algorithm implementations and evaluation metrics described in the following subsections.

Figure 4.1: *A module graph extracted for a sample Java application [1].*



Our second case study is the migration from Java 8 to Java 9, where 10,436 packages of the JDK are clustered into modules. Figure 4.1 depicts a so-called *module graph* extracted for a sample application module (*com.foo.app*) [1]. Each node of this graph represents a cluster (named a module) that contains a set of packages. Each edge represents a dependency of a set of packages within a cluster to a set of packages that are included in another cluster. The contents of each cluster can be derived from the module specification. For instance, a part of the declaration of the *java.sql* module is listed below:

```
module java.sql {
    requires java.logging;
    requires java.xml;
    exports java.sql;
    exports javax.sql;
    exports javax.transaction.xa;
}
```

The module-info.java file in Java 9 outlines the dependencies of the module (requirements) and its provided packages (exports). The dependencies and access permissions across modules are managed in this way by avoiding needless class loading and

improving both security and efficiency. We parsed such module declarations to identify the contents of each cluster. We used the *jdeps* tool[2] to derive dependencies between the packages. We compiled both the clustering of packages in modules and the dependencies among these packages in RSF files. During the process of data collection, it was discovered that certain Java 8 packages were no longer compatible with Java 9 or that certain new packages were only accessible in Java 9. We removed such packages from the dependency files and clustering files to perform a fair comparison between the automatically obtained clustering of Java 8 packages and the clustering of packages introduced in Java 9. Another preprocessing step applied to the input DSM was the addition of a dummy package. Some of the alternative clustering algorithms as listed in the following subsection, require a square matrix where each package has dependencies. However, the JDK includes independent packages. To address this, we added a dummy package and introduced dependencies from these independent packages to the dummy package.

In addition to our model, we used 6 alternative clustering algorithms. These algorithms have been previously proposed/used for software architecture recovery and software modularization. We introduce these algorithms in the following.

4.2 Alternative Clustering Algorithms

We selected 6 algorithms that are employed for software architecture modularization and recovery. In particular, we chose greedy algorithms that are both efficient and effective, as well as frequently used for comparative evaluations in the literature. The selected algorithms are listed below:

- **Bunch NAHC**: Bunch is the first toolset proposed for addressing the software clustering problem [19, 36]. It utilizes hill-climbing algorithms and Next Ascent Hill Climbing (NAHC) is one of these algorithms [36].

[2] `https://docs.oracle.com/en/java/javase/11/tools/jdeps.html`

- **Bunch SAHC**: Steepest Ascent Hill Climbing (SAHC) is another hill-climbing algorithm that is implemented as part of Bunch [36].

- **LIMBO**: scaLable InforMation BOttleneck (LIMBO) [3] is an hierarchical clustering algorithm that utilizes information theory and entropy concepts for software clustering. It is frequently used in comparative evaluations in the literature.

- **ACDC**: Algorithm for Comprehension-Driven Clustering (4 (ACDC) [46] applies pattern-based clustering. It is recognized for its superior performance in previous empirical studies [6].

- **MGMC**: Multi-level Greedy Modularity Clustering (MGMC) [47] is a greedy algorithm that utilizes two heuristics called *greedy coarsening* [48] and *fast greedy refinement* [49]. It has been shown to outperform ACDC on certain benchmark datasets [23].

- **FAST**: Fast Clustering Algorithm [11] is a recently proposed, scalable algorithm for software modularization. Empirical results suggest that it performs better than hierarchical algorithms based on both internal evaluation (modularity achieved) and external evaluation (similarity with respect to the manual grouping/packaging of modules) [11].

The clustering results obtained with these algorithms are evaluated both internally and externally. Internal evaluation measures the extent to which the optimization criteria (i.e., minimizing the total number of imports and reducing redundant package imports) are achieved by the output clusterings. External evaluation assesses the similarity or distance between the output and the ground truth. We describe the set of metrics used to perform these evaluations in the following.

4.3 Evaluation Criteria

We used 4 metrics for evaluation. The first two metrics directly measure to what extent the expectations are met regarding the reduction in the number of imports and the elimination of redundant imports:

- **Number of Clusters Imported (NCI)**: The total number of clusters that are needed to be imported by all the files to satisfy their dependency requirements.
- **Number of Unnecessary Files Imported (NUFI)**: The number of files that are unnecessarily imported along with the imported cluster(s).

In addition to these metrics used for internal evaluation, we used the following 2 metrics for external evaluation, comparing the automatically obtained clusters with respect to those manually created by domain experts:

- **Move or Join eFfectiveness Measure (MoJoFM)**: This similarity metric [50] has consistently been used in empirical studies on software architecture recovery [6, 23, 51, 52] to compare recovered architectures against their ground-truth counterparts. The MoJoFM value for given two clusterings *A* and *B* is calculated as follows:

$$MoJoFM = (1 - \frac{mno(A,B)}{max(mno(\forall A,B))}) \times 100\% \tag{4.1}$$

  Here, $mno(A,B)$ represents the minimum number of *move* or *join* operations required to transform *A* into *B*. Meanwhile, $max(mno(\forall A,B))$ calculates the maximum possible $mno(A,B)$ for any *A*. Higher ***MoJoFM*** values indicate greater similarity between *A* and *B*, whereas lower values signify greater disparity.

- **Architecture to Architecture (a2a)**: Similar to MoJoFM, this metric has also consistently been used in empirical studies on software architecture recovery to

compare recovered architectures against their ground-truth counterparts. a2a [53] is a distance-based measurement, which is defined as:

$$a2a = (1 - \frac{mto(A, B)}{aco(A) + aco(B)}) \times 100\% \tag{4.2}$$

where $mto(A, B)$ is the minimum number of operations needed to transform A into B, $aco(A)$ is the number of operations needed to construct A from scratch.

We present and discuss the results in the following section.

5. RESULTS AND DISCUSSION

This section presents and analyzes the results obtained for both experimental objects. We begin by discussing the findings for the Android TV software library, followed by those for the JDK. The evaluation is structured in two parts: first, we assess the results using internal metrics (NCI, NUFI), and then we examine them using external metrics (MoJoFM, a2a) for both experimental objects. Finally, we conclude the section with an overall discussion of the results and findings. Hereby, we also present additional results on the iterative application of the approach to define the Pareto frontier in the industrial case study.

5.1 Android TV Software Library Organization

Table 5.1 lists the results regarding the comparison of automatically obtained clustering solutions with respect to the one manually created by the domain experts for the Android TV software library. LIMBO requires the user to input the number of clusters. We tested LIMBO for a range of values around the number of clusters suggested by manual clustering and reported the best results achieved. In this case, the bests result were obtained with 40 clusters, as indicated by the corresponding column heading. Overall, the results suggest that the proposed approach achieves the closest clustering to the manual clustering, with the highest scores in the MoJoFM (47.66) and a2a (57.23) metrics.

Table 5.2 lists NCI and NUFI values for the clustering of the Android TV library.

Table 5.1: *The comparison of automatically obtained clustering solutions with respect to the manually created solution for the Android TV software library.*

Method / Metric	ACDC	MGMC	Bunch NAHC	Bunch SAHC	LIMBO (40)	FAST	Our Model
MoJoFM	14.06	10.94	32.03	12.50	25.78	44.53	47.66
a2a	19.68	18.75	39.38	20.13	45.01	52.69	57.23

Table 5.2: *NCI and NUFI values for the clustering of the Android TV software library.*

Method	NCI	NUFI
Manual	218	193
ACDC	151	9,973
MGMC	155	4,524
Bunch (NAHC)	175	2,132
Bunch (SAHC)	168	3,129
LIMBO (40)	281	735
FAST	193	3,129
Our Model	198	0

Figure 5.1: *NCI vs NUFI values obtained for alternative clusterings of the for the Android TV software library.*

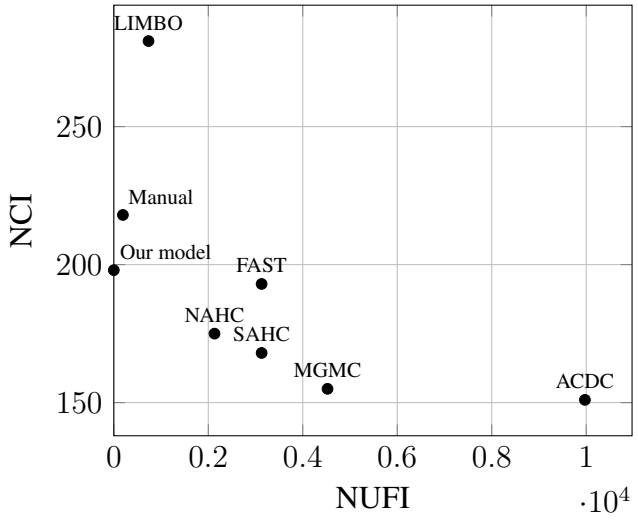


Table 5.3: *The comparison of automatically obtained clustering solutions with respect to the manually created solution for the JDK library.*

Method / Metric	ACDC	MGMC	Bunch NAHC	Bunch SAHC	LIMBO (75)	FAST	Our Model
MoJoFM	70.76	33.29	47.17	57.49	20.76	58.55	64.37
a2a	43.36	34.05	30.96	42.54	12.90	30.66	38.66

The results show that the proposed model significantly reduces unnecessary imports. The proposed model is the only method with a NUFI score of 0, optimizing the modularization of the software system by completely eliminating unnecessary imports. Although the proposed model could not achieve the lowest NCI score, it is seen that it provides a good balance between reducing unnecessary imports and preserving cluster dependencies. The manual clustering has 218 NCI and 193 NUFI scores. This shows that even in the solution created by experts, there are unnecessary dependencies. The results of Android TV library as depicted in Figure 5.1 show that the proposed model provides a structure that is largely similar to clusters created by experts, while significantly reducing unnecessary imports.

5.2 JDK Library Organization

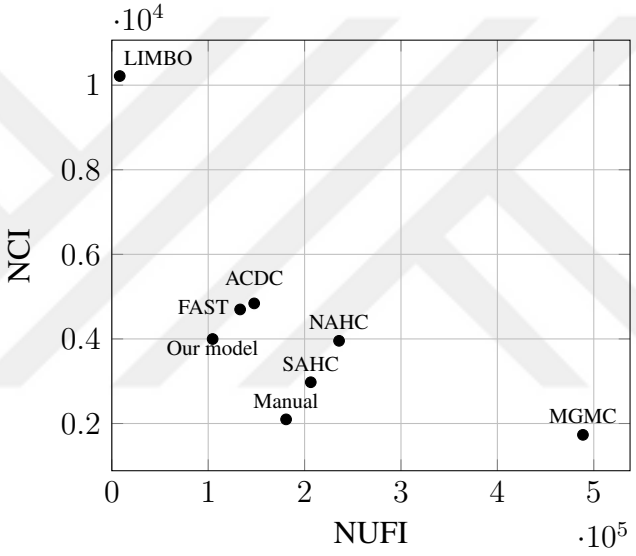
Table 5.3 shows the comparison of the automatically obtained clustering solutions for Java libraries with manual clustering. ACDC evaluating metric produced the results closest to manual clustering by getting the highest scores in MoJoFM (70.76) and a2a (43.56) metrics. Proposed model obtained the the second highest results in MoJoFM (64.37). In the a2a metrics, Bunch (SAHC) is in second with 42.54 while the proposed model achieved the third best result with 38.66.

Table 5.4 lists NCI and NUFI values for the clustering of JDK library. Manual clustering has the lowest NCI value (2,098), while the proposed model has an NCI value of 3,998. This shows that the proposed model creates more clusters compared to the manual solution. In terms of NUFI, the proposed model achieves the lowest score of

Table 5.4: *NCI and NUFI values for the clustering of the JDK library.*

Method	NCI	NUFI
Manual	2,098	180,816
ACDC	4,840	147,714
MGMC	1,732	488,766
Bunch (NAHC)	3,955	235,754
Bunch (SAHC)	2,976	206,410
LIMBO (500)	10,216	8,222
FAST	4,697	133,130
Our Model	3,998	104,603

Figure 5.2: *NCI vs NUFI values obtained for alternative clusterings for the JDK library.*



104,603, second only to LIMBO, which produces a solution with a NUFI below 10,000 but with an NCI exceeding 10,000. The overall results are depicted in Figure B.3. We discuss the obtained results in the following.

5.3 Discussion

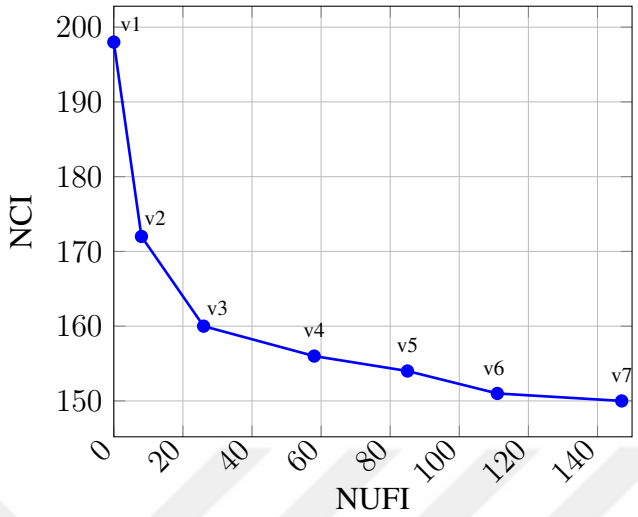
Overall, our findings indicate that the proposed optimization model outperforms other algorithms in terms of internal metrics. This is expected, as the objective function directly incorporates these metrics. The results from the external evaluation are also competitive, reaching or closely approaching the highest observed values. However, some

algorithms produce clusterings that more closely resemble the manually derived decomposition of the JDK library. It is important to emphasize, however, that the manual decomposition does not necessarily represent the ground truth or the optimal solution. Moreover, resemblance to the manual decomposition alone is not a sufficient criterion for evaluation.

For the JDK library, the best results in terms of external metrics are obtained using ACDC. Nevertheless, the clustering produced by our model outperforms ACDC with respect to both NUCI and NUFI (see Table 5.4). Our manual analysis further revealed that the clustering generated by our model is largely aligned with the module organization in JPMS. We observed that our model leads to a more fine-grained clustering, which is also consistent with the results obtained for internal metrics. The number of modules need to be imported almost doubles (increases from 2,098 to 3,998), while the number of packages that are unnecessarily imported decreases by 42% (from 180,816 to 104,603). In the solution proposed by our model, we examined one cluster as an example, which contained 126 packages, all belonging to the *java.base* module. This module actually includes 160 packages in total, with the remaining 34 packages distributed across two other clusters produced by our model. These clusters mainly contain packages under *jdk.internal.** and various security related packages (e.g., *com.sun.security.*, sun.security.**). Thus, the *java.base* module in the JDK is distributed across three clusters in the solution proposed by the model. It is possible to generate alternative clusterings that favor more coarse-grained groupings, as discussed in the following.

Our approach offers flexibility, enabling the creation of alternative clustering strategies that can support software architects in making informed decisions by explicitly considering trade-offs. Notably, our model successfully identified a clustering solution for the Android TV library, where NUFI is 0. Moreover, it is possible to balance an increase in NUFI with improvements in NCI, demonstrating the model's flexibility. Alternative solutions can be explored by adjusting the trade-off between NCI and NUFI. Figure 5.3

Figure 5.3: *Trade-off between NCI vs NUFI depicted for various solutions obtained for the Android TV library organization.*



illustrates several alternative solutions obtained for the Android TV library, where the Pareto front can be analyzed to systematically select the most suitable decomposition. Hereby, the upper bound for the number of redundant imports (NUFI), initially set to 0, is gradually relaxed to reduce the number of clusters by allowing an increasing number of redundant imports. We observe that NCI can be significantly improved without sacrificing much of NUFI in the first few alternative solutions generated. Discrepancies between the selected solution and the original or manually created decomposition may highlight areas for refinement and improvement, depending on the chosen trade-off.

Our experimental evaluation is subject to certain threats to validity. A primary threat to external validity concerns the generalizability of our findings, as the study is based on only two experimental objects. However, these objects are notably diverse: one is a large open-source development kit, while the other is a library used in an embedded system. This diversity helps mitigate concerns about the representativeness of our results. Regarding construct validity, we minimized potential biases by employing multiple clustering algorithms from the literature and evaluating results using a diverse set of well-established internal and external metrics. Additionally, a threat to internal validity

arises from the computational complexity of the problem, which is known to be NP-hard. Nevertheless, our model has demonstrated scalability for libraries containing hundreds of packages, which is sufficient within the scope of our study, as we focus on clustering libraries at a coarse granularity rather than clustering large-scale software systems at the function or class level.

6. CONCLUSIONS

We introduced a set of optimization criteria, models and an optimization approach for modularizing software libraries that are generally organized and clustered based on common usage patterns. Organizing packages of a library into higher-level encapsulation units, such as Java modules, increase the level of abstraction. On the other hand, importing unnecessary packages bundled within a group is undesirable. Our approach balances these concerns. We introduced an hierarchical optimization approach to address this multi-criteria optimization problem.

We evaluated the effectiveness of our model using both an industrial case and the open-source Java SE platform. We employed two internal and two external metrics to evaluate and compare the quality of the obtained results with respect to the manually obtained clustering and results obtained with 6 alternative software modularization techniques. The results show that our approach is highly effective in minimizing both the total number of imports and the number of unnecessary packages imported. It can suggest improvements with respect to both of these criteria at the same time and it allows the designer to make informed trade-off decisions along the Pareto front. The clustering solution generated by our model is not an exact match to the manually created groupings, but it aligns closely while also suggesting improvements and refinements based on the defined criteria.

REFERENCES

[1] M. Reinhold, "The state of the module system," tech. rep., Oracle, 2016. http://openjdk.java.net/projects/jigsaw/spec/sotms/.

[2] I. Sarhan, B. Ahmed, M. Bures, and K. Zamli, "Software module clustering: An in-depth literature analysis," *IEEE Transactions on Software Engineering*, vol. 48, no. 6, pp. 1905–1928, 2022.

[3] P. Andritsos and V. Tzerpos, "Information-theoretic software clustering," *IEEE Transactions on Software Engineering*, vol. 31, no. 2, pp. 150–165, 2005.

[4] J. Garcia, I. Krka, C. Mattmann, and N. Medvidovic, "Obtaining ground-truth software architectures," in *Proceedings of the International Conference on Software Engineering*, pp. 901–910, 2013.

[5] M. Altinisik and H. Sozer, "Automated procedure clustering for reverse engineering PL/SQL programs," in *Proceedings of the 31st ACM Symposium on Applied Computing*, pp. 1440–1445, 2016.

[6] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidovic, and R. Kroeger, "Measuring the impact of code dependencies on software architecture recovery techniques," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 159–181, 2018.

[7] Y. Cai, L. Xiao, R. Kazman, R. Mo, and Q. Feng, "Design rule spaces: A new model for representing and analyzing software architecture," *IEEE Transactions on Software Engineering*, vol. 45, no. 7, pp. 657–682, 2019.

[8] D. Sas and P. Avgeriou, "An architectural technical debt index based on machine learning and architectural smells," *IEEE Transactions on Software Engineering*, vol. 49, no. 8, pp. 4169–4195, 2023.

[9] F. Fontana, R. Roveda, M. Zanoni, C. Raibulet, and R. Capilla, "An experience report on detecting and repairing software architecture erosion," in *Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture*, pp. 21–30, 2016.

[10] M. Altinisik, E. Ersoy, and H. Sozer, "Evaluating software architecture erosion for PL/SQL programs," in *Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings*, pp. 159–165, ACM, 2017.

[11] N. Teymourian, H. Izadkhah, and A. Isazadeh, "A fast clustering algorithm for modularization of large-scale software systems," *IEEE Transactions on Software Engineering*, vol. 48, no. 4, pp. 1451–1462, 2022.

[12] D. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[13] E. Ersoy, K. Kaya, M. Altinisik, and H. Sözer, "Using hypergraph clustering for software architecture reconstruction of data-tier software," in *Proceedings of the 10th European Conference on Software Architecture*, pp. 326–333, 2016.

[14] E. Ersoy and H. Sözer, "Effort estimation for architectural refactoring of data tier software," in *Proceedings of the 19th IEEE International Conference on Software Architecture*, pp. 80–89, 2022.

[15] B. Pourasghar, H. Izadkhah, and M. Akhtari, "Beyond cohesion and coupling: Integrating control flow in software modularization process for better code comprehensibility," *ACM Transactions on Software Engineering and Methodology*, 2024.

[16] K. Jezek, J. Dietrich, and P. Brada, "How java APIs break – an empirical study," *Information and Software Technology*, vol. 65, pp. 129–146, 2015.

[17] S. Ducasse and D. Pollet, "Software architecture reconstruction: A process-oriented taxonomy," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 573 – 591, 2009.

[18] H. Sozer, "Towards a unified approach for software architecture recovery," in *Proceedings of the 48th IEEE Annual Computers, Software, and Applications Conference*, pp. 1316–1317, 2024.

[19] B. Mitchell and S. Mancoridis, "On the evaluation of the Bunch search-based software modularization algorithm," *Soft Computing*, vol. 12, no. 1, pp. 77–93, 2008.

[20] A. Tan, C. Chong, and A. Aleti, "E-SC4R: Explaining software clustering for remodularisation," *Journal of Systems and Software*, vol. 186, p. 111162, 2022.

[21] O. Maqbool and H. Babri, "The weighted combined algorithm: A linkage algorithm for software clustering," in *Proceedings of the 8th Euromicro Working Conference on Software Maintenance and Reengineering*, pp. 15–24, 2004.

[22] P. Andritsos, P. Tsaparas, R. Miller, and K. Sevcik, "LIMBO: Scalable clustering of categorical data," in *Proceedings of the 9th International Conference on Extending DataBase Technology*, pp. 531–532, 2004.

[23] H. Sözer, "Evaluating the effectiveness of multi-level greedy modularity clustering for software architecture recovery," in *Proceedings of the 13th European Conference on Software Architecture*, vol. 11681, pp. 71–87, 2019.

[24] D. Doval, A. Mancoridis, and B. Mitchell, "Automatic clustering of software systems using a genetic algorithm," in *Proceedings of the 9th International Workshop Software Technology and Engineering Practice*, pp. 73–81, 1999.

[25] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, and A. Ouni, "Many-objective software remodularization using NSGA-III," *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 3, pp. 1–45, 2015.

[26] M. Akbari and H. Izadkhah, "Hybrid of genetic algorithm and krill herd for software clustering problem," in *Proceedings of the 5th Conference on Knowledge Based Engineering and Innovation*, pp. 565–570, 2019.

[27] M. Elyasi, M. Simitcioglu, A. Saydemir, A. Ekici, and H. Sozer, "HYGAR: A hybrid genetic algorithm for software architecture recovery," in *Proceedings of the 37th ACM Symposium on Applied Computing*, pp. 1417–1424, 2022.

[28] T. Varol, M. Elyasi, T. Aktas, O. Ozener, and H. Sozer, "Parallelization of genetic algorithms for software architecture recovery," *Automated Software Engineering*, vol. 32, no. 9, pp. 1–24, 2025.

[29] S. Eppinger and T. Browning, *Design Structure Matrix Methods and Applications*. Cambridge, MA, USA: MIT Press, 2012.

[30] A. Kuhn, S. Ducasse, and T. Gîrba, "Semantic clustering: Identifying topics in source code," *Information and Software Technology*, vol. 49, no. 3, pp. 230–243, 2007.

[31] M. Altinisik, H. Sozer, and G. Gursun, "Software architecture recovery from multiple dependency models," in *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*, pp. 1185–1192, 2024.

[32] A. Corazza, S. D. Martino, V. Maggio, and G. Scanniello, "Investigating the use of lexical information for software system clustering," in *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, pp. 35–44, 2011.

[33] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai, "Enhancing architectural recovery using concerns," in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pp. 552–555, 2011.

[34] A. Saydemir, M. Simitcioglu, and H. Sozer, "On the use of evolutionary coupling for software architecture recovery," in *Proceedings of the 15th Turkish National Software Engineering Symposium*, pp. 1–6, 2021.

[35] F. Meng, Y. Wang, C. Chong, H. Yu, and Z. Zhu, "Evolution-aware constraint derivation approach for software remodularization," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, 2024.

[36] B. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the Bunch tool," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 193 – 208, 2006.

[37] K. Praditwong, M. Harman, and X. Yao, "Software module clustering as a multi-objective search problem," *IEEE Transactions on Software Engineering*, vol. 37, no. 02, pp. 264–282, 2011.

[38] N. S. Jalali, H. Izadkhah, and S. Lofti, "Multi-objective search-based software modularization: structural and non-structural features," *Soft Computing*, vol. 23, no. 21, pp. 11141–11165, 2019.

[39] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley Professional, 1994.

[40] E. Triantaphyllou, B. Shu, S. N. Sanchez, and T. Ray, "Multi-criteria decision making: an operations research approach," *Encyclopedia of electrical and electronics engineering*, vol. 15, no. 1998, pp. 175–186, 1998.

[41] A. Ghane-Kanafi and E. Khorram, "A new scalarization method for finding the efficient frontier in non-convex multi-objective problems," *Applied Mathematical Modelling*, vol. 39, no. 23, pp. 7483–7498, 2015.

[42] H. Sozer and C. Gebizli, "Model-based testing of digital TVs: An industry-as-laboratory approach," *Software Quality Journal*, vol. 25, no. 4, pp. 1185–1202, 2017.

[43] M.-A. Storey, K. Wong, and H. Muller, "Rigi: A visualization environment for reverse engineering," in *Proceedings of the 19th International Conference on Software Engineering*, pp. 606–607, 1997.

[44] K. Wong, *RIGI User's Manual*. University of Victoria, 1996.

[45] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using dependency models to manage complex software architecture," in *Proceedings of the 20th Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 167–176, 2005.

[46] V. Tzerpos and R. Holt, "ACDC: An algorithm for comprehension-driven clustering," in *Proceedings of the 7th Working Conference on Reverse Engineering*, pp. 258–267, 2000.

[47] A. Noack and R. Rotta, "Multi-level algorithms for modularity clustering," in *Proceedings of the 8th International Symposium on Experimental Algorithms*, pp. 257–268, 2009.

[48] A. Clauset, M. Newman, and C. Moore, "Finding community structure in very large networks," *Physical Review E*, vol. 70, p. 066111, 2004.

[49] P. Schuetz and A. Caflisch, "Efficient modularity optimization by multistep greedy algorithm and vertex mover refinement," *Physical Review E*, vol. 77, p. 046112, 2008.

[50] Z. Wen and V. Tzerpos, "An effectiveness measure for software clustering algorithms," in *Proceedings of the 12th IEEE International Workshop on Program Comprehension*, pp. 194–203, 2004.

[51] K. Kobayashi, M. Kamimura, K. Kato, K. Yano, and A. Matsuo, "Feature-gathering dependency-based software clustering using dedication and modularity," in *Proceedings of the 28th IEEE International Conference on Software Maintenance*, pp. 462–471, 2012.

[52] J. Garcia, I. Ivkovic, and N. Medvidovic, "A comparative analysis of software architecture recovery techniques," in *Proceedings of the 28th International Conference on Automated Software Engineering*, pp. 486–496, 2013.

[53] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic, "An empirical study of architectural change in open-source software systems," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 235–245, 2015.

APPENDICES

APPENDIX A. JAVA 8 PACKAGE LIST

com.oracle.net
javax.xml.crypto.dsig.*
com.sun.java.util.*
com.sun.jndi.cosnaming
com.sun.media.sound
com.sun.naming.internal
com.sun.java_cup.internal.*
com.sun.jmx.defaults
com.sun.jmx.remote.*
com.sun.rowset.internal
com.sun.security.sasl
com.sun.tracing.dtrace
java.awt.datatransfer
java.awt.dnd
java.nio.file.attribute
java.nio.channels
java.lang
java.lang.instrument
java.rmi.activation
java.rmi.dgc
java.rmi.server
java.time.temporal
java.util.jar
java.util.regex
javax.activation
javax.imageio.event
javax.imageio.plugins.jpeg
javax.annotation
javax.lang.model
java.rmi.registry
javax.net.ssl
javax.security.cert
javax.xml.bind.util
javax.swing.event
javax.management.openmbean
javax.swing.plaf
javax.script
javax.swing.text.rtf
javax.sound.sampled.spi
javax.swing.tree

com.sun.accessibility.internal.*
com.sun.management.jmx
com.sun.image.codec.*
com.sun.jndi.ldap.*
com.sun.org.apache.*
com.sun.jndi.dns
com.sun.imageio.plugins.*
com.sun.jmx.interceptor
sun.applet
com.sun.security.auth.*
com.sun.security.sasl.*
com.sun.xml.internal.*
java.awt.dnd.peer
java.awt.geom
java.beans
java.net
java.nio.charset.spi
java.nio.file.spi
java.awt.peer
java.sql
java.time.chrono
java.security.spec
java.util.logging
java.util.spi
javax.activity
javax.imageio.metadata
java.security.acl
java.security.cert
javax.imageio.stream
javax.xml.bind.helpers
javax.lang.model.util
javax.smartcardio
javax.rmi.ssl
javax.management.modelmbean
javax.swing.filechooser
javax.sound.midi
javax.management.remote
javax.xml.transform
javax.management.remote.rmi
javax.security.auth

com.sun.beans.decoder
com.sun.imageio.spi
com.sun.jmx.snmp.*
com.sun.jndi.url.*
com.sun.net.ssl.*
com.sun.java.browser.*
com.sun.java.swing
com.sun.jmx.mbeanserver
javax.security.sasl
com.sun.org.omg.*
com.sun.swing.internal.*
sun.reflect.generics.parser
java.awt.event
java.awt.image
java.beans.beancontext
java.io
java.nio.file
java.lang.invoke
java.security.interfaces
java.text
java.time.format
java.security
java.time.zone
java.util.zip
javax.annotation.processing
javax.imageio.plugins.bmp
java.util.concurrent
javax.jws
java.text.spi
javax.lang.model.element
javax.management
javax.xml.datatype
javax.swing.colorchooser
javax.swing.plaf.basic
javax.transaction
javax.swing.plaf.metal
javax.swing.plaf.multi
javax.xml.transform.dom
javax.xml.bind.annotation.*
javax.security.auth.callback

com.sun.corba.se.*
com.sun.demo.jvmti.*
sun.awt.geom
com.sun.management
com.sun.nio.file
com.sun.jndi.rmi.*
com.sun.java.swing.*
com.sun.net.httpserver
com.sun.org.glassfish.*
com.sun.rowset.providers
com.sun.tracing
java.awt.color
java.awt.font
java.awt.image.renderable
java.math
java.nio.charset
java.lang.annotation
java.rmi
java.lang.management
java.nio
java.util.prefs
java.util.concurrent.locks
java.util
javax.accessibility
javax.imageio
java.util.stream
java.util.concurrent.atomic
javax.jws.soap
java.time
javax.lang.model.type
javax.security.auth.x500
javax.xml.transform.sax
javax.management.loading
javax.management.monitor
javax.management.relation
javax.xml.bind.annotation
javax.swing.plaf.nimbus
javax.sound.midi.spi
javax.xml.namespace
javax.xml.crypto.dsig

javax.sql
javax.sql.rowset.serial
javax.xml.crypto
javax.print.attribute
javax.tools
javax.naming
javax.xml
javax.swing
javax.swing.text.html.*
javax.xml.transform.stream
com.sun.net.httpserver.*
javax.xml.ws.spi
javax.xml.xpath
com.oracle.util
com.oracle.webservices.internal.*
org.omg.PortableInterceptor
org.omg.SendingContext
org.w3c.dom.ls
org.w3c.dom.css
org.w3c.dom.xpath
sun.applet.resources
sun.awt.datatransfer
sun.awt.dnd
sun.awt.image
sun.dc.pr
com.sun.jndi.ldap
java.nio.channels.spi
sun.invoke.empty
java.util.function
com.sun.beans.infos
com.sun.security.jgss
sun.java2d.jules
sun.java2d.opengl
org.w3c.dom.stylesheets
sun.launcher.resources
sun.management.counter.perf
com.sun.jndi.toolkit.*
sun.management.resources
javax.xml.ws.handler.*
sun.awt.shell
sun.net.ftp.impl
sun.net.spi
sun.net.spi.nameservice
javax.swing.plaf.synth
javax.sql.rowset.spi
javax.xml.stream.events
javax.print.attribute.standard
javax.rmi
javax.xml.bind.attachment
javax.security.auth.spi
javax.swing.text.html
javax.naming.ldap
javax.xml.validation
javax.imageio.spi
javax.xml.ws.spi.*
jdk.internal.util
com.sun.beans.editors
com.sun.activation.registries
org.omg.PortableServer
org.omg.IOP.CodecPackage
org.w3c.dom.ranges
jdk.management.resource.internal
sun.security.krb5.internal.*
org.xml.sax.ext
jdk.internal.util.xml
sun.awt.X11
sun.awt.motif
sun.font
sun.awt.image.codec
javax.xml.soap
sun.management.counter
com.sun.beans.finder
com.sun.beans.util
sun.net.www.content.*
sun.java2d.pipe
sun.java2d.pisces
sun.java2d.xr
sun.management
sun.management.jdp
com.sun.istack.internal
sun.management.snmp
sun.management.snmp.util
jdk.internal.cmm
sun.net.idn
org.omg.CORBA.portable
sun.net.util
javax.management.timer
javax.security.auth.login
javax.print
javax.print.event
javax.rmi.CORBA
javax.naming.directory
javax.naming.event
javax.xml.bind
javax.sound.sampled
javax.xml.ws
javax.xml.ws.http
jdk.internal.org.xml.*
jdk.management.resource
org.omg.CORBA.ORBPackage
org.omg.IOP
com.sun.security.auth
org.w3c.dom.events
com.sun.nio.sctp
com.sun.beans
java.lang.reflect
sun.audio
jdk.internal.util.xml.*
sun.awt.im
sun.awt.windows
sun.awt.util
java.awt.print
sun.instrument
sun.invoke
javax.security.auth.kerberos
javax.xml.transform.stax
sun.java2d.cmm.lcms
sun.java2d.loops
sun.java2d.x11
sun.java2d.cmm
org.omg.CosNaming
sun.management.jmxremote
com.sun.istack.internal.*
sun.management.snmp.jvminstr
jdk.internal.org.objectweb.*
jdk.internal.instrumentation
sun.net.sdp
org.omg.CORBA_2_3
sun.misc.resources
javax.sql.rowset
javax.xml.stream
javax.swing.undo
javax.xml.stream.util
javax.swing.table
javax.transaction.xa
javax.swing.text
javax.swing.border
javax.naming.spi
javax.xml.ws.handler
javax.xml.ws.soap
javax.xml.ws.wsaddressing
com.oracle.nio
com.sun.security.cert.*
org.omg.DynamicAny.*
com.sun.net.ssl
org.w3c.dom.html
jdk.xml.internal
javax.xml.parsers
org.xml.sax
sun.awt
jdk.management.cmm
com.sun.rowset
sun.awt.resources
org.xml.sax.helpers
java.lang.ref
sun.java2d
sun.invoke.util
javax.xml.crypto.dom
javax.net
org.omg.CORBA.*
sun.java2d.pipe.hw
sun.net.www.protocol.*
sun.launcher
java.applet
com.sun.imageio.stream
com.sun.jmx.snmp
sun.management.snmp.jvmmib
jdk
sun.misc
sun.net.smtp
org.omg.CORBA_2_3.portable
sun.net

sun.net.www
sun.net.www.http
sun.nio.ch
sun.nio.cs
sun.print
com.oracle.xmlns.internal.*
sun.awt.event
org.omg.CORBA
sun.reflect.generics.tree
org.omg.PortableInterceptor.*
java.awt
sun.rmi.runtime
sun.rmi.transport.proxy
sun.security.jgss.krb5
sun.security.jgss.spi
com.sun.rmi.rmid
sun.dc.path
java.awt.im.spi
org.w3c.dom.bootstrap
sun.security.pkcs12
sun.security.rsa
sun.security.tools.keytool
sun.security.x509
sun.swing.plaf.synth
sun.swing.text.html
sun.text.resources
sun.tracing
sun.util.calendar
sun.util.logging
sun.util.spi
sun.net.dns
com.sun.awt
org.omg.CosNaming.NamingContextPackage
sun.nio.ch.sctp
sun.print.resources
jdk.management.resource.internal.*
jdk.net
org.omg.CORBA.*
org.omg.PortableServer.CurrentPackage
org.omg.PortableServer.*
sun.reflect.misc
sun.rmi.transport
sun.rmi.transport.tcp
org.omg.PortableServer.portable
sun.security.krb5.internal
sun.security.jgss.spnego
org.omg.stub.java.*
org.omg.stub.javax.*
org.w3c.dom.views
sun.security.provider
sun.security.smartcardio
sun.security.tools.policytool
sun.swing
sun.swing.plaf.windows
sun.text
sun.text.resources.en
sun.tracing.dtrace
sun.util.cldr
sun.util.logging.resources
sun.util.xml
sun.net.ftp
org.omg.CosNaming.*
org.omg.Dynamic
org.omg.DynamicAny
sun.reflect.annotation
sun.net.httpserver
org.jcp.xml.dsig.*
sun.reflect.generics.repository
org.omg.PortableServer.*
org.omg.PortableServer.*
sun.rmi.log
sun.rmi.server
sun.security.action
sun.security.jgss.wrapper
sun.reflect
sun.corba
org.omg.DynamicAny.*
org.w3c.dom.traversal
sun.security.pkcs
sun.security.provider.certpath
sun.security.timestamp
sun.security.util
sun.swing.icon
sun.swing.table
sun.text.bidi
sun.tools.jar
sun.usagetracker
sun.util.locale
sun.util.resources
com.sun.security.ntlm
sun.nio
org.ietf.jgss
sun.nio.fs
sun.reflect.generics.factory
org.omg.IOP.*
sun.reflect.generics.*
sun.reflect.generics.scope
org.omg.Messaging
sun.reflect.generics.visitor
sun.rmi.registry
sun.security.acl
sun.security.jgss
sun.security.krb5
sun.security.jca
sun.dc
java.awt.im
org.w3c.dom
sun.security.pkcs10
sun.security.provider.certpath.*
sun.security.tools
sun.security.validator
sun.swing.plaf
sun.swing.text
sun.text.normalizer
sun.tools.jar.resources
sun.util
sun.util.locale.provider
sun.util.resources.en

APPENDIX B. JAVA 9 MODULE SYSTEM DEPENDENCY GRAPH

Java Standard Edition (JavaSE) Dependency Graph

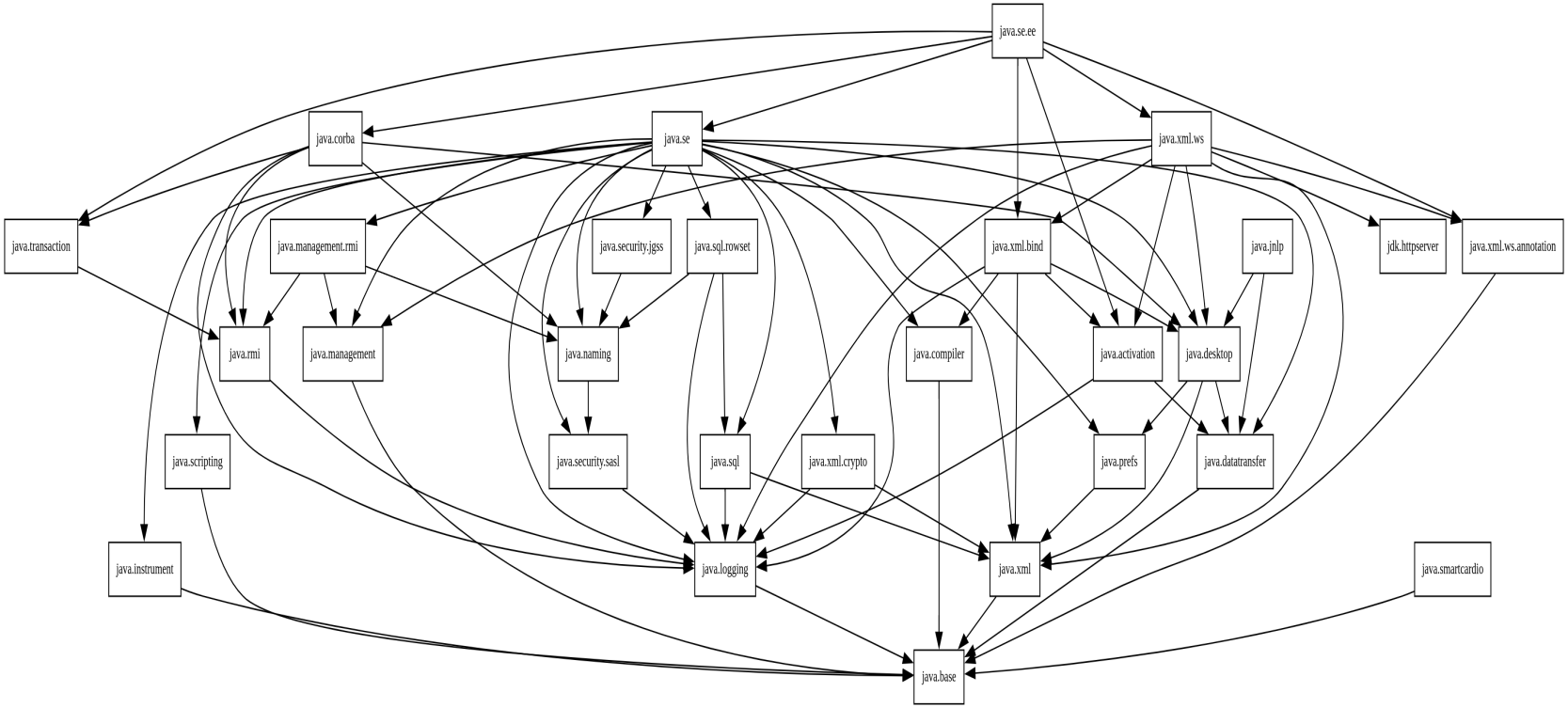


Figure B.1: *JavaSE Dependency Graph*

JavaFX Dependency Graph

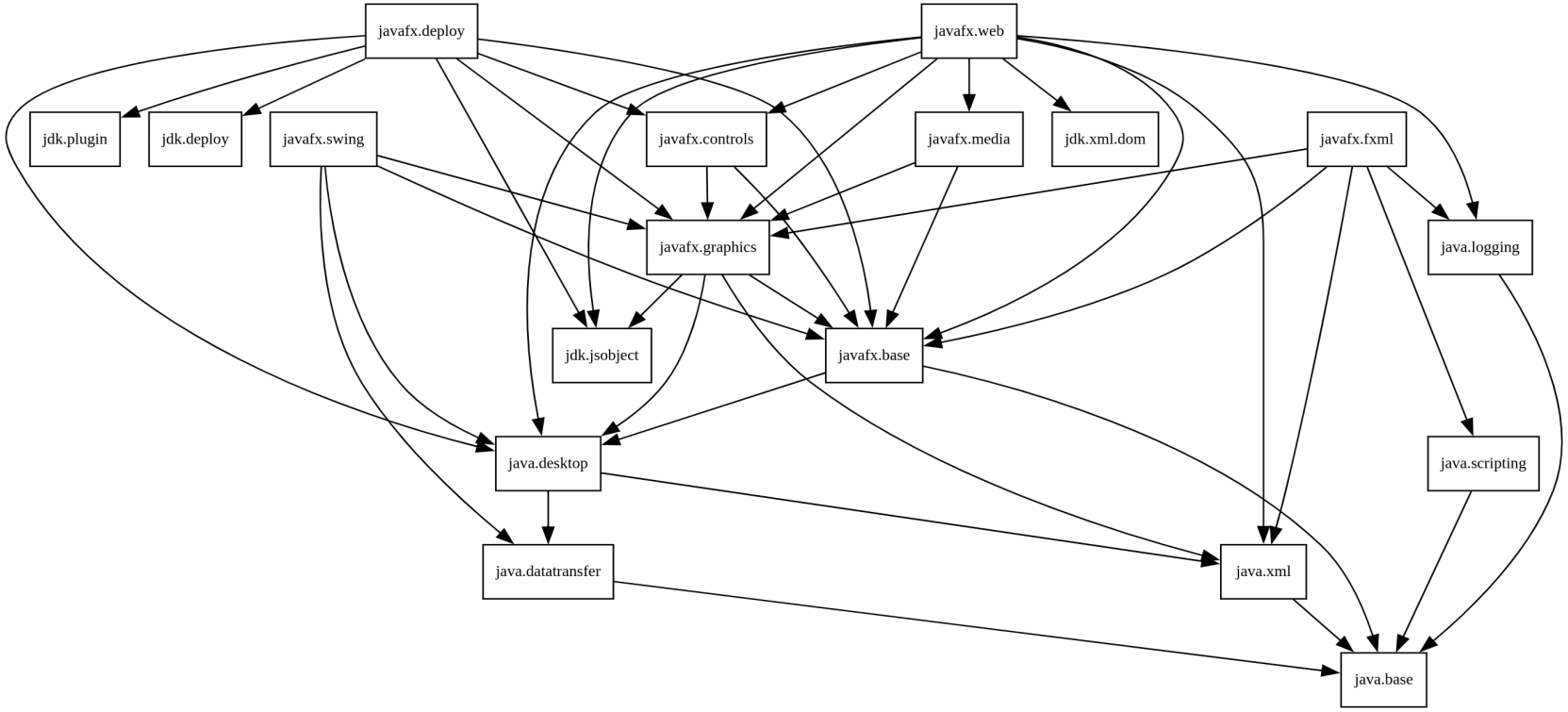


Figure B.2: *JavaFX Dependency Graph*

Java Development Kit (JDK) Dependency Graph

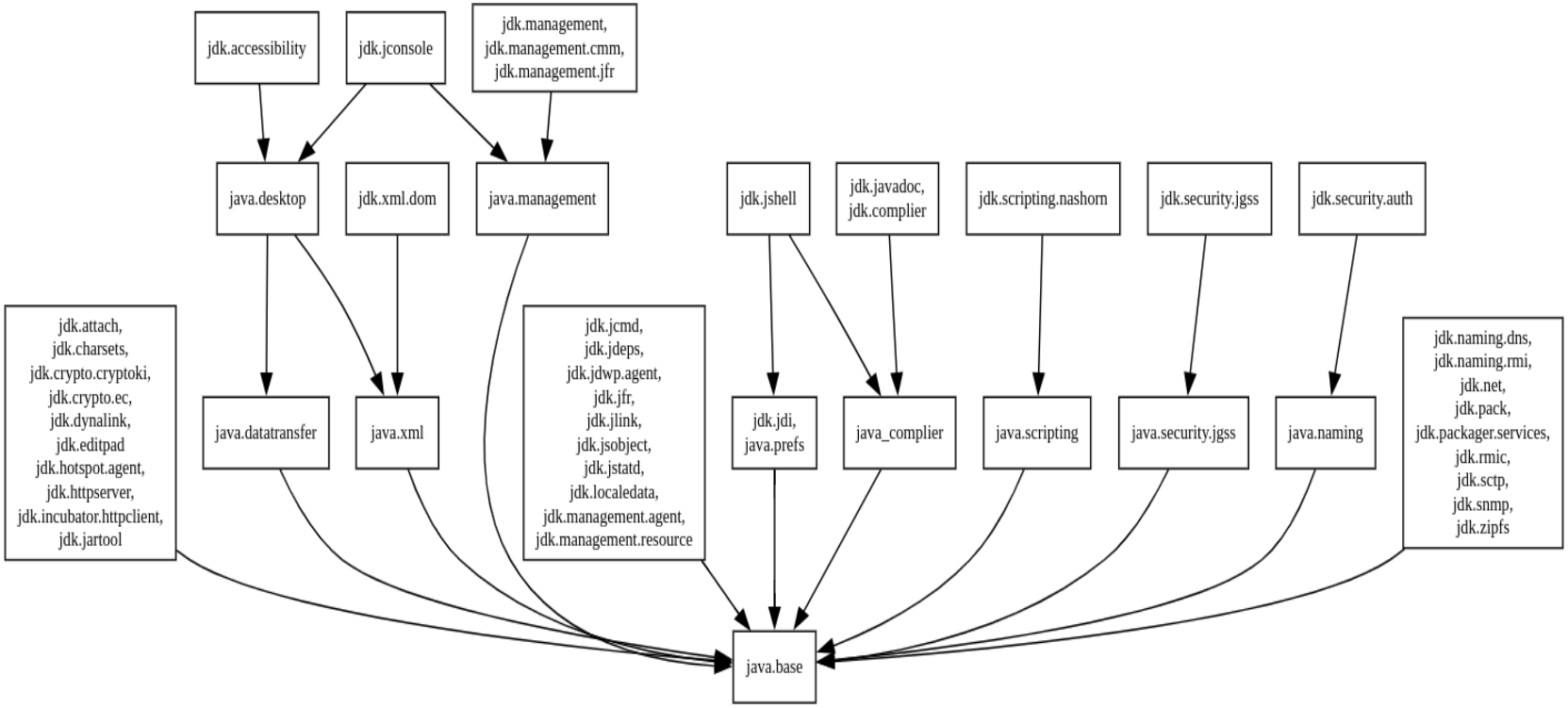


Figure B.3: *JDK Dependency Graph*