Automatic Identification of Cohesive Structures within Modularity Reengineering

Anja Bog Oleksandr Panchenko Kai Spichale Alexander Zeier
Hasso Plattner Institute for Software Systems Engineering
University of Potsdam
August-Bebel-Str. 88, 14482 Potsdam, Germany
Email: {anja.bog, panchenko, kai.spichale, zeier}@hpi.uni-potsdam.de

Abstract—**The quality of software systems depends heavily on the quality of their structure, which affects maintainability and readability. To improve the quality of structure, a system can be restructured. This paper describes a restructuring process, which uses a combination of strongly connected component analysis, dominance analysis, and intra-modular similarity clustering to identify and preserve structures that have been thoughtfully placed together, but would be separated by pure metric-based or similarity-based techniques. The use of the proposed method allows a significant reduction of the number of components that should be moved. Therefore, the number of false movements is alleviated. The proposed approach was implemented in a prototype and illustrated by statistics and examples from 18 open source Java projects. A coherence metric is introduced to further improve restructuring results.**

Keywords-**Source code organization, Restructuring, reverse engineering, and reengineering, Metrics**

I. Introduction

Each time that a software element (e.g., method, class, package) is added, the developer has to decide where this element has to be placed. It is likely that the developer chooses a suboptimal position because of the limited ability of humans to cope with the increasing complexity of software systems. Besides this, any source code change could introduce new dependencies among software elements, which might adversely affect the system structure. Dependencies could also vanish, which allows creating simplified configurations. This paper is an extended version of our previous work [1], integrating the proposed pre-processing techniques into the entire process of restructuring and adding discussions about further techniques to enhance the results, i.e., cohesion.

In this paper we present an approach for generating restructuring advice to improve the physical structure [2] of software systems. Restructuring advice comprises moving misplaced software elements, whereas dependencies among software elements are kept unchanged. Thus, restructuring advice leads to another system configuration.

The proposed approach includes a preprocessing phase and a restructuring phase. In the restructuring phase, several alternative configurations of the original system are created and compared to each other based on coupling, cohesion, and coherence. However, not all configurations that lead to better values of these metrics are acceptable. Not heeding design decisions of the original system and only improving metric values, may pull apart cohesive structures consisting of elements that were thoughtfully placed together. Therefore, given configurations must not be ignored as they capture well-considered design decisions. The preprocessing phase identifies such structures that should be preserved during restructuring and helps to distinguish between intended and unaware decisions.

Restructuring advice is created as the result of the preprocessing and restructuring phase. In the following steps this advice is validated by developers and eligible restructuring advice can be implemented. For the preprocessing phase we propose techniques that are applied to identify intended cohesive structures and to mark them for preservation during restructuring. Since the techniques used in this paper are based purely on structural analysis of the software system, semantical meanings of the elements are not taken into account and are out of scope for this paper. The results of the preprocessing phase are further enhanced by applying the cohesion metric in the restructuring phase in order to choose an optimal system configuration.

The following section introduces the graph structure used as the basis for the restructuring algorithms. Section III relates the approach to existing research. An overview of the proposed reengineering process is given in Section IV. The subsequent section focuses on the preprocessing phase of the reengineering process detailing the steps and algorithms, which are applied in order to create restructuring advice. Section VI discusses a further possibility to improve the restructuring results by introducing coherence metric. Afterwards, a short overview of our implementation to create restructuring advice is given in Section VII. Section VIII concludes the paper and gives an overview of possible directions for future work.

II. Module Dependency Graph

The proposed restructuring approach is independent of any programming language. To accomplish this objective, the described techniques are based on the Module Dependency Graph (MDG) [3] that has three types of elements: *components*, *modules*, and *dependencies*. A *component* represents an atomic software element whose internal structure is not considered at this level of granularity. Calls between components are represented by dependencies. Each pair of distinct components can be linked by at most one *dependency*. The components and their dependencies form a directed graph. *Modules* are disjoint sets of components. Figure 1 shows the elements of an MDG. Notice that the inter-modular dependency between b and c implies a module dependency between M_1 and M_2.

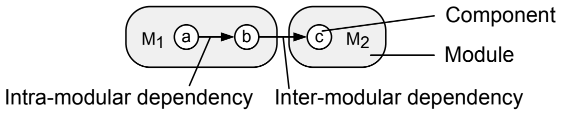


Figure 1. MDG Elements

A graph is defined as a pair (V, E), where V is the vertex set and E is the edge set. However, the graph described above must be extended to satisfy the need to reflect modularization properties. Therefore a partition is defined. A partition of a set X is a set of nonempty subsets of X such that every element x in X is in exactly one of these subsets. Thus, an MDG is a triple (V, E, M), where (V, E) is a directed graph and M represents modules and is a partition of V.

The MDG can be applied at different levels of granularity. Java applications can be modeled as follows: Java classes are represented by *components* and packages by *modules*. Experiments have also been executed on a larger system developed with the SAP [4] component model that divides software projects into development components (DCs) to organize the software in comprehensible and reusable units. Further, a software component (SC) combines DCs to larger units for delivery and deployment. The DCs are modeled by *components* and SCs by *modules*.

Traditionally, various metrics have been used to assess the quality of the MDG. Most popular metrics for coupling and cohesion are used as optimization criteria for metric-based refactoring. Coupling of a module *m* [5, p. 520] is the degree of dependence between *m* and other modules of the MDG and is represented by the number of afferent and efferent dependencies. Cohesion of the module *m* [5, p. 524] is the measure of the strength of structural connections of components inside *m* and is calculated as the number of actual dependencies divided by the number of maximal possible dependencies within a module. The module, that has only one component, has a cohesion value equal to one.

III. Related Work

Several techniques for automating the decomposition of software systems into subsystems and improving their structure exist. In this section, categories of techniques, concrete techniques, and their fields of application are presented. Tool-driven reengineering techniques are aimed at architecture reconstruction and software restructuring. Architecture reconstruction captures component recovery and program understanding of single systems and product lines [6]. Software restructuring aims at improving the physical design of existing code [7]. Due to the high number of software elements and relations among them, maintaining and improving the structural quality must be supported and automated by tools. As we are interested in improving the structure of software systems, we are focusing on software restructuring techniques in the following. Design structure matrices [8] and reflexion models [2] provide means to model the structure of software systems and thereby gain insights to support their maintenance and evolution. Furthermore, (semi-)automated techniques exist, that extract abstractions from software artifacts to make software systems more understandable, e.g., Storey et al. [9] developed the interactive, visual tool Rigi that helps understanding software systems.

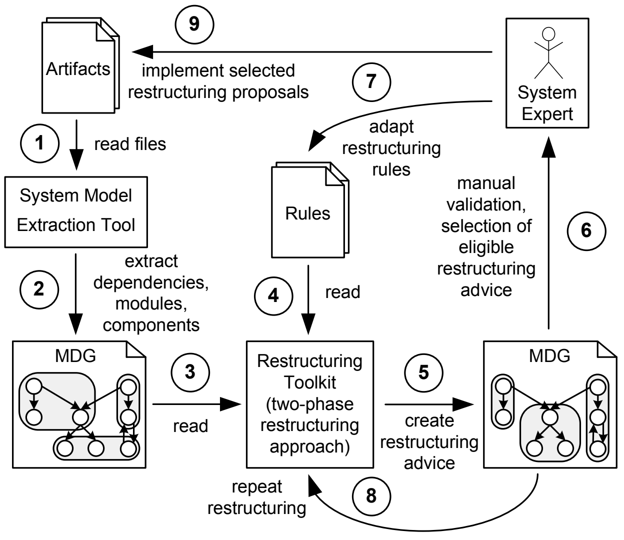
Beyond modeling the structure of a software system, subsystem decomposition techniques help to provide proposals for improving its structure. Respective techniques are categorized by their underlying technologies into clustering techniques, graph-based techniques, and multi-approach techniques [10]. Clustering techniques utilize similarity measures for components and modules to group the most similar ones. Graph-based techniques model relevant properties of subsystems as graph properties and optimize them. Multi-approach techniques use a mixture of techniques from the aforementioned categories.

Concerning clustering techniques, Hutchens and Basili [7] proposed an algorithm that clusters procedures by measuring the interaction between pairs of procedures. Schwanke's tool Arch [11] clusters similar software elements based on their common and distinct references. Girard, Koschke, and Schied [12] extended Schwanke's similarity metric to cluster functions, types, and variables into atomic elements.

K-cut modularization proposed by Jermaine [13] is an example for a graph-based technique. This method decomposes a software system into modules in such a way as to attempt the minimization of inter-module connections. As a result, modules with high cohesion and low coupling are identified. The problem of software decomposition is formulated as the k-cut problem in graph theory. The computation of the k-cut of a graph is an NP-hard problem, however, efficient approximations exist [14]. K-cut modularization is most appropriate for monolithic procedural systems.

Regarding multi-approach techniques, Mitchell and Mancoridis [15] developed Bunch, a tool that identifies subsystems based on maximizing cluster cohesion, while minimizing inter-cluster coupling. Tzerpos and Holt [16] developed the algorithm ACDC that recognizes subsystem patterns and places software elements based on lowering coupling.

However, none of the mentioned techniques was explicitly developed for providing restructuring advice for misplaced components. There is no technique that detects subsystem patterns to preserve existing structures. ACDC detects subsystem patterns to create a skeleton, but the



Artifacts
9
implement selected restructuring proposals
System Expert
7
adapt restructuring rules
1
read files
System Model Extraction Tool
Rules
manual validation, selection of eligible restructuring advice
6
2
extract dependencies, modules, components
4
read
MDG
3
read
Restructuring Toolkit (two-phase restructuring approach)
5
create restructuring advice
MDG
repeat restructuring
8

Figure 2. Reengineering Process

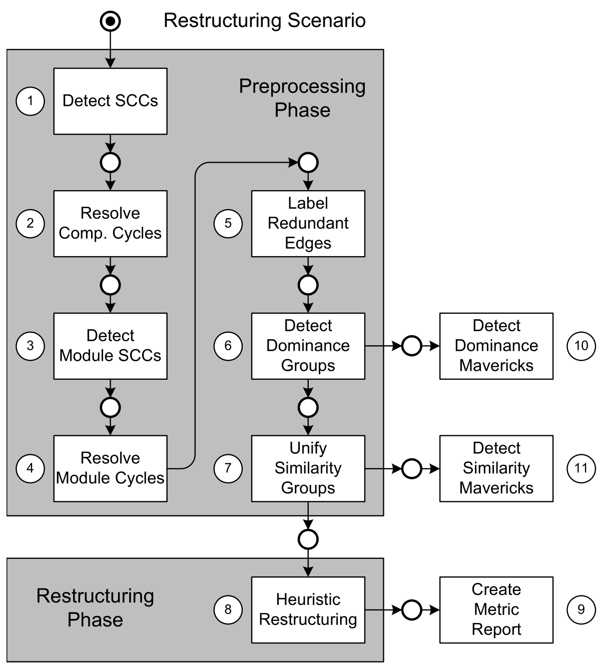


Figure 3. The Two-phase Restructuring Approach

identified subsystems are not restricted to the given configuration. Furthermore, a subsystem decomposition technique is needed that also resolves cyclic module dependencies. This need is based on the Acyclic Dependency Principle, a design principle formulated by Robert C. Martin [17] that indicates that dependencies between software elements of the granularity of release must not form cycles. Applied to the MDG, this design principle stipulates that any cyclic dependencies between modules have to be resolved. Our empirical investigation shows that, although this design principle is widely accepted, the most systems lack of proper structure.

Techniques that are merely based on maximizing cluster cohesion and minimizing inter-cluster coupling cannot create acceptable results because the proposed configuration often requires too many component moves. High cohesion and low coupling are commonly agreed to be attributes of good design. Although, a configuration with optimal metric values does not inevitably imply an optimal design. Furthermore, clustering techniques based solely on similarity cannot reasonably place all software elements because of too low similarity values.

In the following section we will introduce our reengineering process that detects and preserves constructs that have been consciously placed together.

IV. Reengineering Process

Figure 2 shows the steps and their sequence within a simplified version of the entire reengineering process. (1) The process starts with analyzing the physical artifacts (e.g., source code, deployment descriptors, configuration files) of a software system. (2) Tools automatically extract data about the software elements and the dependencies among them to create a system model in the form of an MDG. (3) The created MDG is the input data for the used restructuring techniques. (4) Rules can be defined to limit possible restructuring proposals that contradict intended design decisions. Such rules comprise modification suppressions for software elements. By this means a component can be bound to a module in such a way that it cannot be moved into another module. (5) Finally, graph theory and clustering techniques are applied to propose restructuring advice. (6) Not all proposals might be suitable to the intended design. Therefore the proposals must be validated and selected by a system expert. (7) If the restructuring proposals are not satisfying, the rules can be adapted. (8) After changing the rules, the analysis can be repeated. (9) The process is finished when the approved restructuring proposals are implemented.

The detailed activities of the two-phase restructuring approach step that is proposed in this paper are shown in Figure 3. The restructuring approach comprises a number of individual graph-based techniques as well as clustering techniques that are applied one after another to the MDG. The selection and order of these techniques depend on the intended purpose, which is in our case restructuring of an existing software system in order to improve the overall structure.

(1) In the first step, all cyclic dependencies on component level are detected in the form of strongly connected components (SCCs). One separate or several overlapping dependency cycles constitute a strongly connected component. Cyclic dependencies are useful information for restructuring insofar as placing all components of an SCC into the

same module reduces the inter-modular coupling. (2) In the second step, the detected component SCCs are collapsed and placed into appropriate modules. Collapsing SCCs pulls together all interconnected components. By this means the dependency structure becomes acyclic at component level. Further, cyclic module dependencies are resolved if the inter-connected components were originally in different modules. (3) Next, cyclic module dependencies are detected. Even if the dependency graph is acyclic on component level after collapsing all SCCs, cyclic module dependencies can exist. Cyclic dependencies among modules are architectural flaws that have to be detected and removed. (4) Cyclic module dependencies are resolved by moving components from one module to another. (5) For dominance analysis, redundant edges have to be labeled. (6) Dominance subgraphs can then be detected and collapsed without introducing new cyclic dependencies. (7) In the last step of the preprocessing phase, similar components are unified by clustering them within the same module.

(8) In the second phase, component moves between different modules are proposed by heuristic restructuring in order to improve metric values. (9) At the end a metric report is created that compares the original configuration with the proposed configuration based on metrics.

(10) Detection of dominance mavericks shows misplaced components based on dominance analysis. The results of dominance mavericks detection are not integrated into the final restructuring advice as the number of false positives is high according to our experiments. Nevertheless, these results should be reviewed by an expert of the analyzed system and integrated manually if applicable. (11) Similarity mavericks detection analyzes components that are misplaced based on similarity. Results are not part of the final restructuring advice, either, but should be reviewed by an expert as they might reveal further structural improvement.

In the following section, more detail about the steps within the preprocessing phase is provided.

V. Preprocessing Phase

The purpose of the preprocessing phase is to (1) resolve cyclic module dependencies and to (2) identify cohesive structures with dominance analysis and intra-modular similarity clustering.

Removing cyclic dependencies in a software system increases maintainability and extensibility as will be explained in this section. Additionally, acyclic graphs are a prerequisite for the dominance analysis in the following step.

The goal of identifying cohesive structures is to distinguish between thoughtfully intended and unaware decisions to position components in order to improve the final reengineering results. Dominance analysis detects connected components, and similarity clustering identifies elements with similar structure.

Table I
Analyzed projects

Name	Source
Apache Ant 1.7.1	http://ant.apache.org/
CruiseControl 2.7.3	http://cruisecontrol.sourceforge.net/
Eclipse Ganymede SR1	http://www.eclipse.org/ganymede/
Apache Geronimo 2.1.2	http://geronimo.apache.org/
Hibernate 3.3.0	http://www.hibernate.org/
JBoss 4.2.3 jdk6	http://www.jboss.org/
JDepend 2.9	http://clarkware.com/software/JDepend.html
J2SE 5.0 JDK 1.5.0_09	http://java.sun.com/javase/
JRuby 1.1.4	http://jruby.codehaus.org/
JUnit 4.5	http://www.junit.org/
Apache Logging Services for Java 1.2.15	http://logging.apache.org/
Apache Maven 2.0.9	http://maven.apache.org/
NetBeans 6.1 (Base IDE)	http://www.netbeans.org/
PicoContainer 2.5.1	http://www.picocontainer.org/
Saxon 9.1.0.1	http://saxon.sourceforge.net/
Spring Framework 2.5.5	http://www.springsource.org/
Apache Tomcat 6.0.18	http://tomcat.apache.org/
Xalan-j 2.7.1	http://xml.apache.org/xalan-j/

The examples given in this paper are selected after performing an analysis of 18 open source Java projects. The usefulness of the approach is exemplified by statistics. The list of the selected projects is given in Table I.

To automate the analysis, a tool has been implemented. The tool uses Classycle[18] to extract runtime dependencies among Java classes. JAR files are analyzed by Classycle and, as a result, the MDG is created in XML format. The algorithms used in the proposed approach have been implemented to work with the MDG in this format.

A. Resolving Cyclic Module Dependencies

Cyclic dependencies form SCCs. An SCC of a digraph G is a maximal strongly connected subdigraph of G. A digraph is strongly connected if there is a directed walk from each vertex to each other vertex [19].

The components of an SCC can be part of several modules. If this is the case, cyclic module dependencies are created. To resolve these cyclic module dependencies, the SCCs are collapsed. Possible locations of a collapsed SCC are the modules that contain at least one component that is part of the SCC. The module that implies the lowest coupling is chosen.

Even if the dependency graph is acyclic on component level after collapsing all SCCs, cyclic module dependencies can exist. Alternative modifications to remove these pseudo-cyclic dependencies are *component moving*, *module splitting*, and *module merging*.

As the name states, in component moving a component is moved from one module to another to resolve the pseudo-cycle. Component moving may result in an empty module, which has to be deleted. In module splitting a selected module is split into two separate modules in such a way

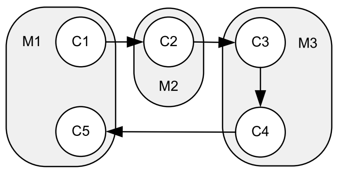


Figure 4. Pseudo-cyclic Dependencies on Component Level

that the pseudo-cyclic dependencies are removed. However, not every module that is part of the cycle can be split so that the intended effect occurs. Module splitting can be interpreted as a special case of component moving, as it requires the movement of components into a newly created or existing module. Consequently, the conditions under which this strategy is applied are the same as in component moving. Module merging is the reverse modification of module splitting. Modules are unified with the objective of turning their undesired inter-modular dependencies into intra-modular dependencies. Module merging is not selective, i.e., modules are merged in their entirety. Therefore, component moving and module splitting are far more fine-grained than merging.

To determine which components are part of a cycle on module level that is not cyclic on component level, the order of the modules has to be determined and which components are the cause of it. Each module in a cycle has the role of a successor and predecessor to another module. Therefore, for each module pair (p, s) two component sets called predecessor subset P and successor subset S are defined, where p is the predecessor and s the successor. "$\rightarrow$" denotes dependencies between components. The sets are defined as follows:

$$\begin{aligned}P(p,s) &:= \{v \mid v \in p \wedge \exists t \in s : \\ &\quad \exists\{r_1, ..., r_n\} \subseteq p : v \rightarrow r_1 \rightarrow ... \rightarrow r_n \rightarrow t\} \\ S(p,s) &:= \{v \mid v \in s \wedge \exists r \in p : \\ &\quad \exists\{t_1, ..., t_n\} \subseteq s : r \rightarrow t_1 \rightarrow ... \rightarrow t_n \rightarrow v\}\end{aligned}$$

The predecessor subset contains all components $v \in p$ that directly or indirectly depend on a component $t \in s$ not considering paths including other modules. Similarily, the successor subset is the subset of s containing components v that are directly or indirectly referenced by the components in p.

The algorithm for resolving pseudo-cyclic dependencies is based on the fact that the inter-component dependencies are acyclic. Consequently, there exists at least on non-empty set of components that can be moved to resolve the cyclic dependencies on module level.

Figure 4 shows an example containing pseudo-cyclic dependencies on module level. To determine which component subsets can be moved to resolve these dependencies, predecessor subset and successor subset are analyzed for each of the modules, see Table II.

If the predecessor subset of a module overlaps with its successor subset, then neither the predecessor nor the successor subset can be moved to break the pseudo-cyclic dependencies. In the example, only C_1 and C_5 could effectively be moved. Hence, we can identify the following options:

- Moving $\{C_1\}$: The component subset $\{C_1\}$ is the predecessor subset of M_1 with regard to M_2. If $\{C_1\}$ is moved to M_2, the dependencies on module level become acyclic.
- Moving $\{C_5\}$: The successor subset of M_1 with regard to M_3 is $\{C_5\}$. Moving this subset to M_3 is another valid solution.
- Splitting M_1: One of the identified disjoint subsets can be moved into a separate module.

If multiple solutions exist, the solution is chosen that requires the smallest number of component moves and creates the configuration with the lowest coupling. Typically, software systems contain a large number of cycles [20]. If several cycles overlap, the algorithm has to be applied iteratively.

According to Fowler [21], cycles in dependency structures should be avoided as they provoke situations, where every change of one module breeds other changes that come back to the original module entering a vicious circle of change propagation. Systems become tightly coupled by cyclic dependencies and fiercely resist decomposition.

Drawbacks of cyclic dependencies are: (1) higher complexity, since modules cannot be understood independently. The goal of modularization is to divide a complex system into simpler modules that can be independently developed, maintained, and understood [22], whereas tight coupling, caused by cyclic dependencies diminishes the ability to understand modules in isolation [23, p. 85]; (2) less flexibility and extensibility is a result of cyclic dependencies as the program is harder to understand because of increased complexity, and coupled components can be affected by changes. Cycles make it harder to accurately assess and manage the impact of changes to the system.

Cyclic dependency analysis is an important aspect of the proposed approach because 31.5% of the components and 53.7% of the modules in the analyzed projects are involved in cyclic dependencies. Melton and Tempero's empirical study [20] confirms the high amount of cyclic dependencies between classes and packages, which was also discovered in our analysis: 52% of the component level SCCs remain inside a module. Consequently 48% of the component level SCCs are distributed over more than one module and cause cyclic dependencies among modules.

A large number of cyclic dependencies requires many component movements to resolve cycles, which results in complex refactorings at the beginning of the process. In

Module	Predecessor subset	Successor subset	Intersection of subsets
M_1	$P(M_1, M_2) = \{C_1\}$	$S(M_3, M_1) = \{C_5\}$	$\emptyset$
M_2	$P(M_2, M_3) = \{C_2\}$	$S(M_1, M_2) = \{C_2\}$	$\{C_2\}$
M_3	$P(M_3, M_1) = \{C_3, C_4\}$	$S(M_2, M_3) = \{C_3, C_4\}$	$\{C_3, C_4\}$

Table II. Predecessor subset, successor subset and subset intersections of the example given in Figure 4

this case, human intervention is needed to continue with the reengineering process.

B. Dominance Analysis

Components often reference underlying components that provide specific functions, which cannot be understood or reused individually. If an underlying component is an essential part of the referencing component, then referenced and referencing components must not be separated by any restructuring attempt.

Figure 5 (A) shows a client using a facade, a unified interface hiding a complex subsystem. The facade and the covered components must be reckoned as one unit to prevent dispersing this coherent structure.

Figure 5 (B) shows two clients depending on some utility components. When `Client2` was developed, its common utility functions were extracted to the component `CommonUtil`. `Client1` can use `CommonUtil` without referencing `Client2`. The component `SpecialUtil` emerged when the developers of `Client1` decided to encapsulate some functions. But no other component depends on `SpecialUtil`. `Client1` and `SpecialUtil` belong together and must not be separated. Nevertheless, if `SpecialUtil` was developed as a reusable component, a rule could be defined to enable the separation of both components.

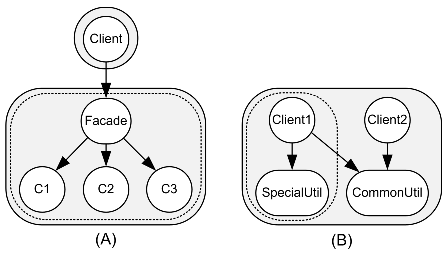


Figure 5. Examples of Dominance Subgraphs

Dominance analysis is the process of identifying intramodular subgraphs that can be collapsed without introducing cyclic dependencies. The first step, is collapsing all SCCs as mentioned above. This step is common to other proposed approaches [24], [25], because the algorithms for transitive closure used for dominance analysis require acyclic graphs as input. Next, all redundant dependencies are removed. An edge e part of a directed graph G is said to be redundant iff e can be removed without changing the transitive closure of G [26]. Then, the algorithm goes through all vertices v and examines whether v qualifies as dominated vertex. The vertex v is said to be dominated iff there exists exactly one vertex d that is linked to v by an edge (d, v). Dominator vertex and dominated vertex form a dominance pair if they are part of the same module. One separate or several overlapping dominance pairs constitute a dominance subgraph. The dominance subgraph detection is repeated until no further dominance pairs can be detected.

Figure 6 shows an example of dominance analysis. The SCC $\{e, f\}$ detected in part (A) is collapsed in part (B). The dotted edges in part (B) denote redundant dependencies. In part (C) the redundant dependencies are filtered and three dominance pairs are found that form two collapsed dominance subgraphs in part (D).

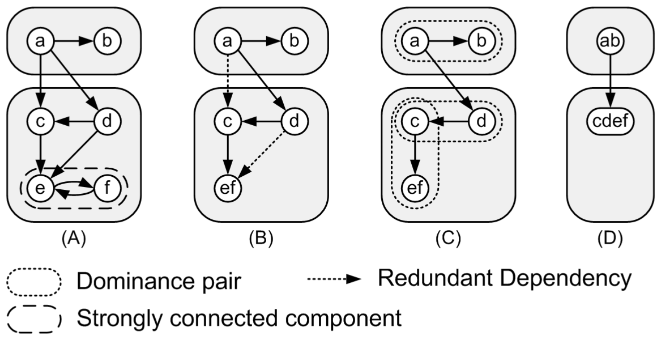


Figure 6. Steps of Dominance Analysis

Only components within the same module should be united, otherwise too many components would be pulled together. Since dominance subgraphs are not spread over multiple modules, subsequent restructuring attempts must either move complete subgraphs or keep them unchanged in their modules.

Other proposed dominance analyses [16], [24] are restricted to rooted (sub-)trees and unsuitable to detect nested dominance subgraphs due to redundant edges.

During preprocessing 32.1% of the analyzed classes could be assigned to dominance subgraphs. There are 1.82 dominance subgraphs per package. Based on a manual review, the identified dominance subgraphs are accurate and expedient without exception. Figure 7 shows an intramodular dominance subgraph detected in the J2SE JDK. The Java classes `Timer`, `TimerThread`, `TaskQueue`, and `TimerTask`, which are part of the `java.util` package, form a dominance subgraph. When the system is restructured these classes should be kept together because `Timer` and

`TimerTask` are always referenced together by classes positioned in other packages. `TimerThread` and `TaskQueue` are only used by `Timer`, and therefore they should not be separated from `Timer`.

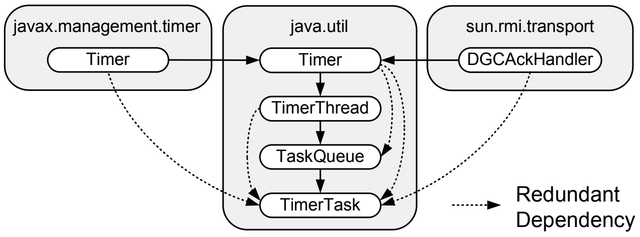


Figure 7. Detected Dominance Subgraph

C. Intra-modular Similarity Clustering

Structural similarity clustering allows comparing components based on afferent and efferent dependencies. Two patterns can be distinguished: support library pattern and facade pattern. Figure 8 (A) shows the support library pattern. The gray component is a support library that is used frequently. Figure 8 (B) shows the facade pattern. The gray component is the facade depending on a number of other components. In both cases, the white components resemble one another structurally although the dependencies of the support library and facade may be irrelevant for positioning. Therefore, it can be useful to remove these dependencies from consideration.

Clustering algorithms [27] group similar entities together. In order to quantify the similarity of entities a similarity measure is necessary. Schwanke [11] proposes a similarity measure to compare two procedures. This measure is applied to the MDG to compare components. By this means clusters of similar components that are part of the same module can be identified. These clusters are cohesive structures that are sustained during restructuring.

Figure 9 shows the similar components b and c that would be separated by metric-based restructuring techniques without similarity clustering. Part (A) shows the initial MDG. The similarity cluster $\{b, c\}$ is marked by a shaded oval. Without this cluster, b an c would be separated to improve metric values as shown in part (B). The metric values for the original configuration are: Coupling(M_1 and M_3) = 2, Coupling(M_2) = 4, Cohesion(M_1 and M_3) = 1, Cohesion(M_2) = 0. The alternative configuration created by a pure metric-based approach would have: Coupling(M_1 and M_3) = 2, Cohesion(M_1 and M_3) = 0.5. Part (C) shows an alternative configuration with equal metric values. Therefore, using structural clusters prevents pulling apart similar components.

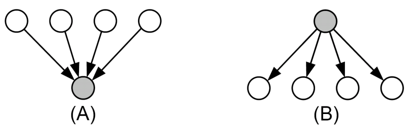


Figure 8. Similarity Clustering Motivating Example

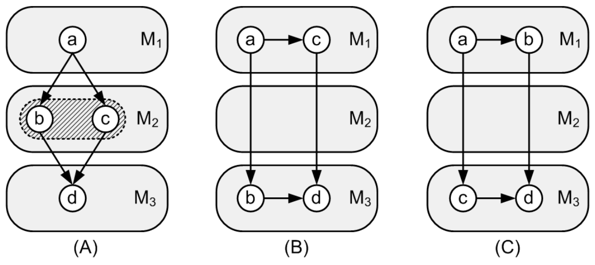


Figure 9. Intra-modular Similarity Clustering

The similarity measure is based on features that are derived from the afferent and efferent dependencies of the components. Let a be a component that depends on the component b, then a has the feature "is-predecessor-of-b" and b has the feature "is-successor-of-a". Important features occur seldom, while common features emerge frequently. For example, the dependencies to a logging component are of little importance because they occur frequently throughout the system. Schwanke proposes to use the Shannon information content [28] from information theory as weighting factor for features. The formula for the weight of a feature used in this project is:

$$weight = -1 * log_2 \frac{\#feature\ references}{\#components - 1}$$

The components are clustered as follows: first an undirected graph is created. Each component is represented by a distinct vertex. At the beginning the graph has no edges. Then pairs of similar vertices are connected if the components they represent are part of the same module and if the similarity value reaches the similarity threshold, which has been detected in experiments as 0.8. At the end, the connected components of the graph are detected. Each connected component represents a cluster of similar components.

The collapsed dominance subgraphs and SCCs can affect the similarity of components. Therefore, similarity must be measured based on graphs without collapsed subgraphs.

The experiments show that only 61.3% of the classes are in the same package as their most similar peer. Therefore, restructuring a system by means of clustering the most similar components causes a high number of component moves and is therefore not acceptable. Seen from a different point of view, the positioning of 38.7% of the classes might be justified by other arguments, which are not detectable by similarity clustering.

Similarity clustering is a useful tool for detecting structures that should be maintained during restructuring. 12.3% of the analyzed classes could be assigned to intra-modular similarity clusters using a high similarity threshold to limit the number of false-positive findings.

VI. Restructuring Phase

Although the main focus of this paper is preserving cohesive structures during the preprocessing phase, the accuracy of the restructuring phase can be improved as well. Besides using well known and widely accepted coupling and cohesion metrics, this paper introduces a third metric – coherence – as a further optimization criterion.

A. Coherence Metric Definition

Cohesion refers to the relatedness of a module's internal structure. We argue that an external viewpoint should also be used to analyze how the elements of a module contribute to a common purpose or objective. Therefore, we propose the metric coherence, which characterizes the functional cohesion of a module from an external viewpoint.

For a module m the function $Clients$ defines the components that are not part of m and that depend on components in m.

$$Clients(m) := \{c | c \in V \wedge c \notin m \wedge \exists a \in m : (c, a) \in E\}$$

Let m be a module and c a component not in m. The function ref specifies the components in m, which are used by c.

$$ref(c, m) := \{a | a \in m \wedge (c, a) \in E\}$$

The Jaccard Coeffcient [29, ch. 7] is used as a binary similarity measure to compare the usage patterns of the module's external clients. Let A and B be sample sets by which two entities are compared, then the Jaccard Coefficient is

$$S_{Jaccard} := \frac{|A \cap B|}{|A \cap B| + |A \triangle B|}$$

with the symmetric difference: $A \triangle B := (A \setminus B) \cup (B \setminus A)$. Coherence for a module m is defined as the sum of Jaccard Coeffcients applied to the module's clients:

$$Coherence(m) := \frac{\sum_c |ref(a, m) \cap ref(b, m)|}{\sum_c |ref(a, m) \cap ref(b, m)| + \sum_c |ref(a, m) \triangle ref(b, m)|}$$

with $c := \{a, b\} \subset Clients(m), a \neq b$. Coherence quantifies the similarity of usage patterns of the module's external clients. All clients of module m are pairwise compared using sets of referenced components in m.

Figure 10 shows three modules with varying coherence. The modules M_1, M_2, and M_3 are equal, but are used differently. Module M_1 has two clients each referencing to a different component in M_1. By means of the above proposed formula $Coherence(M_1) = 0/(0 + 2) = 0$. The value 0 corresponds to our intuitive comprehension of coherence because the clients use disjoint parts of M_1. If all elements of a module would contribute to one and the same purpose or objective, the clients would depend on component subsets with high intersection.

Module M_2 has three clients. `Client1` and `Client3` depend on different components. `Client2` depends on both components from m. In this case, coherence has a low value, but not zero, $Coherence(M_2) = (1+0+1)/((1+0+1)+(2+1+1)) = 1/3$.

Both clients of M_3 have the same usage pattern. In this case the module provides a coherent set of functions to other elements in the system. Consequently, coherence has the highest possible value, $Coherence(M_3) = 2/(2 + 0) = 1$.

A similar idea of using clients of a module for measuring the strength of its internal connections has been used in the Lack of Coherence in Clients (LCIC) metric [30]. LCIC has been used for identifying candidates for refactoring. The main difference between the coherence metric presented in this paper and LCIC is that LCIC uses the same approach as the Lack of Cohesion on Methods (LCOM) metric [31] while the coherence metric is based on a similarity measure. We argue that in contrast to the LCIC our metric depend less on the size of the module.

B. Coherence Metric Properties

This section validates the cohesion metric according to a property set similar to the set of properties proposed by Briand et al. [32] that must be satisfied by coupling and cohesion metrics.

Non-negativity and normalization property requires the existence of a real number Max such that the coherence of a module belongs to an interval $[0; Max]$. The metric coherence has the range $[0; 1]$.

Zero value property requires coherence to be zero, if the usage patterns of clients have nothing in common. Coherence is not defined for modules without clients. If a module has one client, the coherence is 1 per definition. Let us assume a module m has n clients, where $n = |Clients(m)| \geq 1$. If all clients depend on different components in m, then $|\forall a, b \in Clients(m), a \neq b : ref(a, m) \cap ref(b, m) = \emptyset$ and consequently $Coherence(m) = 0$.

Monotonicity property means that the coherence of a module is not decreased by adding an inter-modular dependency between a client and a component that is already

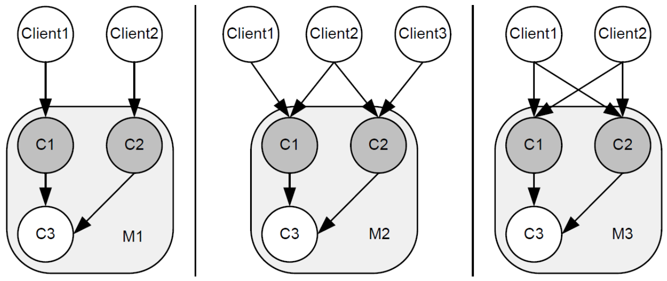


Figure 10. Illustration of Coherence Metric

referenced by one ore more other clients of this module. Let $\Gamma = (V, E, M)$ be an MDG, $m \in M$ a module, and $t \in m$ a component. Let $c_1, c_2 \in Clients(m)$ be two different clients of m. $(c_1, t) \notin E, (c_2, t) \in E, E' := E \cup \{c_1, t)\}$, and $\Gamma' = (V, E', M)$ is a second MDG. This property is satisfied, if the coherence of m in Γ is not greater than the coherence of m in Γ'. If in Γ the coherence of m is a/b with $a \geq 0$ and $b > 0$, then in Γ' the coherence of m is $(a + x)/b$ with $x \geq 1$ because there is at least one more common dependency leading to t. Adding a dependency (c_1, t) increases the similarities of the usage patterns of the clients of m.

Coherent modules property means that the coherence of a module created by merging two other modules having different clients is not greater than the maximum coherence of the two original modules. Let $\Gamma = (V, E, M)$ be an MDG. Let $m_1, m_2 \in M$ be two different, non-empty modules, $m_1 \neq m_2$, $|m_1|, |m_2| > 0$. Further, there is a module $m = m_1 \cup m_2, m \notin M$. Let $M' := M \cup \{m\} \setminus \{m_1, m_2\}$ be a set of modules and $\Gamma' = (V, E, M')$ an MDG. The coherence of m_1 and m_2 is:

$$Coherence(m_1) = \frac{r_1}{r_1 + d_1} \quad Coherence(m_2) = \frac{r_2}{r_2 + d_2}$$

where $r_1, d_1, r_2, d_2 \in \mathbb{N}$ and $r_1 + d_1, r_2 + d_2 > 0$. In order to validate this property we have to show:

$$max\{Coherence(m_1), Coherence(m_2)\} \geq Coherence(m)$$

Let us assume

$$\begin{gathered} Coherence(m_1) \geq Coherence(m_2) \\ \Leftrightarrow \frac{r_1}{r_1 + d_1} \geq \frac{r_2}{r_2 + d_2} \\ \Leftrightarrow r_1(r_2 + d_2) \geq r_2(r_1 + d_1) \end{gathered}$$

Let $x \geq 1$ be the number of usage difference of the clients that were added when merging m_1 and m_2. If $Coherence(m_1) \geq Coherence(m_2)$, then it is sufficient to show that

$$\begin{gathered} Coherence(m_1) \geq Coherence(m) \\ \Leftrightarrow \frac{r_1}{r_1 + d_1} \geq \frac{r_1 + r_2}{r_1 + r_2 + d_1 + d_2 + x} \\ \Leftrightarrow r_1^2 + r_1 r_2 + d_1 r_1 + d_2 r_1 + r_1 x \geq r_1^2 + r_1 r_2 + d_1 r_1 + d_1 r_2 \\ \Leftrightarrow r_1(r_2 + d_2) + r_1 x \geq r_2(r_1 + d_1) \end{gathered}$$

C. Coherence Metric Values

A manual inspection of the coherence metric values confirmed its plausibility. For example, the package `org.apache.tools.ant.taskdefs` is a conglomeration of different, partially related classes. This package has no clearly defined function, but containing all Ant tasks. The coupling is 978, cohesion 0.01, coherence 0.08. Other packages such as `org.apache.tools.tar` include a set of related classes. Its coupling is 8, cohesion is 0.2, and coherence is 0.62.

The Spearman's rank correlation coeffcient, p, was used to measure the pairwise correlation between the module size and coherence, and between cohesion and coherence. There is a very significant (p-$value \ll 0.05$), medium negative correlation between size and coherence (p = –0.52). Further, there is a very significant (p-$value \ll 0.05$), medium positive correlation between cohesion and coherence (p = 0.42).

The distribution of the cohesion and coherence values of the analyzed Ant projects is provided in Figure 11. The coherence metric shows a wider spectrum than cohesion. As a result of this stronger distinction of the projects regarding their coherence value, we argue, that although both metrics correlate, coherence can complement coupling and cohesion as a further optimization criteria.

VII. The Automatic Restructuring Toolkit

As a basis for the validation of our proposition, we developed a framework called "Automatic Restructuring Toolkit" (ART). This section comprises a short description of ART and its characteristic implementation aspects.

ART is a framework providing a collection of individual restructuring techniques, e.g. "detect SCCs" or "resolve component cycles" as introduced above. Figure 12 gives an overview of the most important modules within ART. All artifacts needed to analyze the structure of a software system are provided as XML documents, which are transferred into the internal format by the XML Handler and the Data Loader. The ArtGraph is an MDG that has been created from source code with the help of external tools, e.g., Classycle in case of Java code. The core engine of ART contains the restructuring techniques, which can be combined with each other via tasks to create restructuring proposals according to specific restructuring processes such as the process shown in Figure 3. The output of each task is the input of its following task. Intermediate results are stored, since they can be helpful to understand the final results.

The task-based approach allows changing the execution order of techniques without re-compilations. Therefore, techniques can be added or replaced, and configured with diverse parameters leaving the entire process of restructuring flexible. The composition of techniques can be accomplished through the exposed Java API, or by Ant scripting.

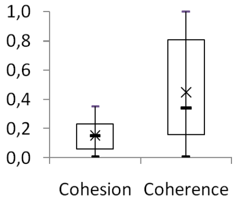


Figure 11. Cohesion and Coherence Metric Values

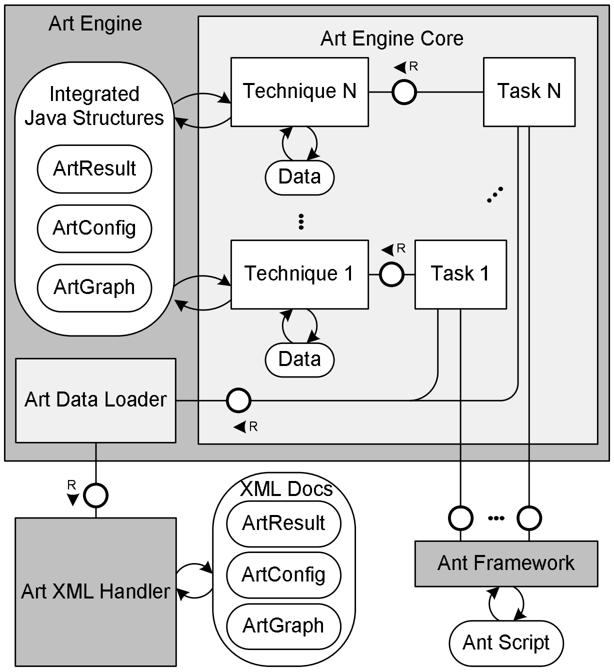


Figure 12. ART Architecture Overview

VIII. Conclusion and Future Work

Instead of radical changes, manageable changes are proposed by the presented approach. Existing cohesive structures are identified in the preprocessing phase and preserved during restructuring.

Eighteen Java open source projects have been analyzed for this work. The analysis shows that each module could be split on an average into 3.1 modules without introducing new inter-modular dependencies. 76% of all dependencies are inter-modular. Consequently, pure metric-based techniques would propose many component moves and split up those modules not changing the values for coupling, but improving the cohesion values.

Since only 61.3% of the components are in the same module with their most similar peer, pure similarity-based techniques would also propose comprehensive changes.

The results verify the usefulness of the proposed approach. During preprocessing 32.1% of the analyzed classes could be assigned to dominance subgraphs and 12.3% could be assigned to similarity clusters for preserving these structures during restructuring, thereby proposing less radical change.

The approach, however, does not include statements about the actual usage during runtime. Cases may exist where the usage patterns implying components to be similar on the basis of a structural analysis seldom or never occur during the runtime of the system. Runtime analysis and validating the above techniques from this point of view is a stream for future work.

More empirical research is necessary to analyze to what extent preserving cohesive structures supports or impedes finding better configurations. Future tests will show whether size and quality of the intra-modular similarity clusters can be improved with an extended similarity measure [12].

In future work the restructuring rules will be extended and combined with logical architectures mapped onto the physical artifacts of the analyzed systems to reduce the level of uncertainty of restructuring proposals.

A similar approach can be used during development of new software to identify positions for a new component while the rest of the system is kept unchanged.

Another field of future work lies in assessing different versions of a software system with our proposed approach, hereby validating the approach and the design decisions made during the evolution of the system.

Although no empirical evidence about the usefulness of the coherence metric has been investigated in this paper, we believe that introducing this additional criteria will allow preserving more cohesive structures. An experiment to test this hypothesis is part of future work.

Our results show that pure structural analysis can significantly contribute to the improvement of source code structure. From our point of view including analysis of the components' semantic meaning may even lead to further enhanced restructuring results. The validation of this assumption is subject of future work.

Acknowledgments

We thank Pieter Bloemendaal and Jakob Spies for their insightful inputs and suggestions concerning this work.

References

[1] K. Spichale, O. Panchenko, A. Bog, and A. Zeier, "Preserving Cohesive Structures for Tool-based Modularity Reengineering," in *Proceedings of the Fourth International Conference on Software Engineering Advances, ICSEA'09*, Porto, Portugal, September 2009.

[2] G. C. Murphy, D. Notkin, and K. J. Sullivan, "Software Reflexion Models: Bridging the Gap between Design and Implementation," *IEEE Transactions on Software Engineering*, vol. 27, no. 4, pp. 364–380, 2001.

[3] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner, "Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures," in *Proceedings of the IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 1999, p. 50.

[4] "SAP – Business Managemant Spftware Solutions Applications and Services," http://www.sap.com, accessed July 1st, 2010.

[5] H. Zuse, *A Framework of Software Measurement*. Hawthorne, NJ, USA: Walter de Gruyter & Co., 1997.

[6] R. Koschke, "Atomic Architectural Component Recovery for Understanding and Evolution," Ph.D. dissertation, University of Stuttgart, 2000.

[7] D. H. Hutchens and V. R. Basili, "System Structure Analysis: Clustering with Data Bindings," *IEEE Transactions on Software Engineering*, vol. 11, no. 8, pp. 749–757, 1985.

[8] S. Huynh, Y. Cai, Y. Song, and K. Sullivan, "Automatic Modularity Conformance Checking," in *Proceedings of the International Conference on Software Engineering*. ACM, 2008, pp. 411–420.

[9] M.-A. D. Storey, K. Wong, H. A. Müller, P. Fong, D. Hooper, and K. Hopkins, "On Designing an Experiment to Evaluate a Reverse Engineering Tool," in *Proceedings of the 3rd Working Conference on Reverse Engineering*. IEEE CS Press, 1996, pp. 31–40.

[10] J.-F. Girard, "ADORE- AR: Software Architecture Reconstruction with Partitioning and Clustering," Ph.D. dissertation, Univ. of Kaiserslautern, CS Dept., 2006.

[11] R. W. Schwanke, "An Intelligent Tool For Re-engineering Software Modularity," in *Proc. of the Int. Conference on Software Engineering*. IEEE CS Press, 1991, pp. 83–92.

[12] J.-F. Girard, R. Koschke, and G. Schied, "A Metric-Based Approach to Detect Abstract Data Types and State Encapsulations," *Automated Software Engineering*, vol. 6, no. 4, pp. 357–386, 1999.

[13] C. Jermaine, "Computing Program Modularizations Using the k-cut Method," in *Proceedings of the 6th Working Conference on Reverse Engineering*. Los Alamitos, CA, USA: IEEE CS Press, 1999, pp. 224–234.

[14] O. Goldschmidt and D. Hochbaum, *Polynomial algorithm for the k-cut problem*. Los Alamitos, CA, USA: IEEE Computer Society, 1988, vol. 0.

[15] B. S. 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.

[16] V. Tzerpos and R. C. Holt, "ACDC: An Algorithm for Comprehension-Driven Clustering," in *Proceedings of the 7th Working Conference on Reverse Engeneering*. IEEE CS Press, 2000, pp. 258–267.

[17] R. C. Martin. (2000) Design Principles and Design Patterns. http://www.objectmentor.com.

[18] "Classycle: Analysing Tools for Java Class and Package Dependencies," http://classycle.sourceforge.net, accessed July 1st, 2010.

[19] J. L. Gross and J. Yellen, *Handbook of Graph Theory*. CRC Press, 2004.

[20] H. Melton and E. Tempero, "An Empirical Study of Cycles among Classes in Java," *Empirical Software Engineering*, vol. 12, no. 4, pp. 389–415, 2007.

[21] M. Fowler, "Reducing Coupling," *IEEE Software*, vol. 18, no. 4, pp. 102–104, 2001.

[22] D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[23] E. Yourdon and L. L. Constantine, *Structured Design: Fundamentals of a discipline of Computer Program and Systems Design*. Raleigh, NC, USA: Prentice-Hall, Inc., 1979.

[24] J.-F. Girard and R. Koschke, "Finding Components in a Hierarchy of Modules: A Step Towards Architectural Understanding," in *Proceedings of the International Conference on Software Maintenance*. IEEE CS Press, 1997, pp. 58–65.

[25] A. Cimitile and G. Visaggio, "Software Salvaging and the Call Dominance Tree," *Journal of Systems and Software*, vol. 28, no. 2, pp. 117–127, 1995.

[26] A. V. Aho, M. R. Garey, and J. D. Ullman, "The Transitive Reduction of a Directed Graph," *SIAM Journal*, vol. 1, no. 2, 1972.

[27] T. A. Wiggerts, "Using Clustering Algorithms in Legacy Systems Remodularization," in *Proceedings of the 4th Working Conference on Reverse Engineering*. IEEE CS Press, 1997, pp. 33–43.

[28] R. G. Gallager, *Information Theory and Reliable Communication*. John Wiley & Sons, Inc., 1968.

[29] M. Falk, F. Marohn, and B. Tewes, *Foundations of Statistical Analyses and Applications with SAS*. Basel, Swiss: Birkhäuser, 2002.

[30] S. Mäkelä and V. Leppänen, "A Software Metric for Coherence of Class Roles in Java Programs," in *Proceedings of the 5th international symposium on Principles and practice of programming in Java*. New York, NY, USA: ACM, 2007, pp. 51–60.

[31] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[32] L. C. Briand, S. Morasca, and V. R. Basili, "Property-Based Software Engineering Measurement," *IEEE Transactions on Software Engineering*, vol. 22, no. 1, pp. 68–86, 1996.