Preserving Cohesive Structures for Tool-based Modularity Reengineering

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

Abstract

The quality of software systems heavily depends on their structure, which affects maintainability and readability. However, the ability of humans to cope with the complexity of large software systems is limited. To support reengineering large software systems, software clustering techniques that maximize module cohesion and minimize inter-modular coupling have been developed. The main drawback of these approaches is that they might pull apart elements that were thoughtfully placed together. This paper describes how strongly connected component analysis, dominance analysis, and intra-modular similarity clustering can be applied to identify and to preserve cohesive structures in order to improve the result of reengineering. The use of the proposed method allows a significant reduction of the number of component movements. As a result, the probability of false component movements is reduced. The proposed approach is illustrated by statistics and examples from 18 open source Java projects.

1. 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.

An approach for generating restructuring advice for improving the physical structure [1] of software systems is presented in this paper. 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. The scope of this paper is the preprocessing phase. We propose techniques that are applied in this phase 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 following section relates the approach to existing research. An overview of the proposed reengineering process is given in Section 3. Section 4 introduces the graph structure used as the basis for the restructuring algorithms. The subsequent section focuses on the preprocessing phase of the reengineering process. It details the steps and algorithms, which are applied in order to create restructuring advice, that can then be selected for implementation. Section 6 concludes the paper and gives an overview of possible directions for future work.

2. Related Work

Design structure matrices [2] and reflexion models [1] can provide insight into the structure of software systems to support their maintenance and evolution. Furthermore, (semi-)automatic subsystem decomposition techniques that extract abstractions from software artifacts to make software systems more understandable exist. Storey et al. [3] developed the interactive, visual tool Rigi that helps understanding

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. However, the identified subsystem patterns have not been used to assess structural quality. There is a need for a subsystem decomposition technique that also resolves cyclic module dependencies.

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, however, a configuration with optimal metric values does not inevitably imply an optimal design.

Furthermore, clustering techniques based on similarity cannot reasonably place all software elements because of too low similarity values.

3. Reengineering Process

Figure 1 shows the proposed 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 a modular dependency graph (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.

4. Modular Dependency Graph

The proposed restructuring approach is independent of any programming language. To accomplish this objective, the described techniques are based on the MDG 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 2 shows the elements of an MDG. Notice that the inter-modular dependency between b and c implies a module dependency between M1 and M2.

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 SAP1 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 $M$ is the degree of dependence between $M$ and other modules of the MDG and is represented by the number of afferent and efferent dependencies [9, p. 520]. Cohesion of the module $M$ 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 [9, p. 524]. The module, that has only one component, has a cohesion value equal to one.

5. 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.

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 1.

To automate the analysis, a tool has been implemented. The tool uses Classycle\(^2\) 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.

5.1. Resolving Cyclic Module Dependencies

Cyclic dependencies form strongly connected components (SCC). 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 [10].

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. Such cycles are resolved by moving components from one module to another. If multiple solutions exist, the solution is chosen that requires the smallest number of component moves and creates the configuration with the lowest coupling.

According to Fowler [11], 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 [12], whereas tight coupling, caused by cyclic dependencies diminishes the ability to understand modules in isolation [13, 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 [14] 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 this case, human intervention is needed to continue with the reengineering process.

5.2. Dominance Analysis

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

Figure 3 (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 3 (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 3. Examples of dominance subgraphs

Dominance analysis is the process of identifying intra-modular 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 [15][16], 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 \( G \) [17]. 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 4 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).

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][6] 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 5 shows an intra-modular 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.

5.3. 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 6 (A) shows the support library pattern. The gray component is a support library that is used frequently. Figure 6 (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 [18] group similar entities together. In order to quantify the similarity of entities a similarity measure is necessary. Schwanke [7] 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 7 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 \(M_2\)) = 2, Coupling(\(M_2\) and \(M_3\)) = 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.

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 [19] from information theory as the 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.

### 6. 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.

18 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 [8].

In future work the restructuring rules will be extended. Additional MDG elements (e.g., layers) can be introduced, to model the analyzed systems more precisely and 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, thereby validating the approach and the design decisions made during the evolution of the system.

Acknowledgment

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

References


