Analysis of Multi-Dimensional Code Couplings

Fabian Beck
VISUS, University of Stuttgart, Germany
Email: fabian.beck@visus.uni-stuttgart.de

Abstract—Software systems consist of hundreds or thousands of files, which are usually not independent of each other but coupled. While it is obvious that structural dependencies like method calls or data accesses create couplings, there also exist other, more indirect forms of coupling that should be considered when modifying, extending, or debugging a system. In contrast to most previous research, in this work, code coupling is considered as a multi-dimensional construct: several forms of structural couplings are contrasted to couplings based on the history and the semantics of the source code entities. The work proposes two novel visualization techniques, which allow for exploring and visually comparing different concepts of coupling. Based on an empirical study on open source systems, the work further provides insights into the relationship between concepts of coupling and the modularization of software; first evidence on the usage of modularization principles can be derived thereof. Finally, a new application for adapting the modularization of a software system—component extraction—is introduced and tested with varying coupling data. This work summarizes the doctoral thesis of the author, suggests directions for future research, and reports lessons learned.

I. INTRODUCTION

If changing one module in a program requires changing another module, then coupling exists.
—Martin Fowler, 2001 [1]

Defining coupling by those hypothetical changes, multifaceted reasons exist why entities of a software system need to be changed together—coupling is indicated by different observable features of the system:

- **Structure**: Changing the functionality or interface of a code entity may induce changing another because the code depends on functionality defined somewhere else.

- **History**: The history of a software project might reveal hidden dependencies: code entities changed together in the past are likely to still change together in the future.

- **Semantics**: When adapting the specification of a software system, all entities that relate to the modified semantics need to be changed, which are not necessarily connected by structure or history.

These observable connections between entities of code are called *code couplings* in this work. The term serves as a superordinate to other terms often used in a similar context, such as *code dependencies*, *relationships*, *associations*, *links*, *similarities*, or *connections*. Each form of observable relationship introduces a different *concept of code coupling*.

Usually, coupling is not treated as a multi-dimensional but one-dimensional construct, often consisting of only structural dependencies. There exist indeed some works that analyze different concepts of coupling, but more on an exemplary level than applying and comparing a larger number of concepts systematically. In contrast, this work is considering code coupling as a multi-dimensional construct and aims at providing the means for systematically analyzing these couplings. Hence, the central goal of this work is to provide a better understanding of different concepts of code coupling in particular context of modularity.

This work summarizes and reflects the doctoral thesis of the author [2], which was conducted under the supervision of Stephan Diehl at the University of Trier, Germany between February 2009 and February 2013. Stephan Diehl and Martin Pinzger (University of Klagenfurt, Austria) acted as examiners for the thesis. The work was part of the DFG project “Analyse mehrdimensionaler Kopplung zur Unterstützung des Software-Entwicklungsprozesses” (grant number: DI 728/8-1/2). The remainder of this paper presents a synopsis of the results of the thesis (Section II), discusses directions for future research (Section III), and reports some lessons learned (Section IV).

II. SYNOPSIS

Formally, code couplings are modeled as a graph structure consisting of code entities as nodes and couplings as weighted, directed edges. Multiple concepts of coupling are treated as different types of edges. The modularization of a system additionally introduces a hierarchy on the nodes. The final data model is a construct called *modularized multi-dimensional coupling graph*, which aggregates, for a single software system, all data relevant in this work. Focusing on object-oriented Java systems, interfaces and classes form the code entities and the package structure provides a hierarchical organization.

The 17 concepts of coupling considered in this work are divided by three categories: *structure*, *history*, and *semantics*. Goal for selecting specific concepts was to cover a broad spectrum of relevant concepts that also could be retrieved with acceptable effort. The specific concepts are listed in Table I: First, the structural couplings of the system are covered by direct inheritance dependencies (*SD.Inh*), aggregation dependencies (*SD.Agg*), and usage dependencies (*SD.Use*); further structural dependencies are considered by indirect variants of these three concepts—code entities are coupled if they depend on the same other project-external (*FO.InhE, FO.AggE, FO.UseE*) or project-internal (*FO.InhI, FO.AggI, FO.UseI*) code entities. Second, history is reflected in two variants of evolutionary coupling recording co-changing code entities in the past (*EC.Conf, EC.Conf*), and two variants of code ownership coupling connecting the files having similar authors (*CO.Bin, CO.Prop*). Third, semantic relations in code are...
expressed through two variants of coupling based on shared code clones of code entities (CC.I, CC.II) and two variants of measuring the similarity of the vocabulary used in source code (SS.Tfidf, SS.LSI).

The thesis falls into three main parts addressing different objectives (Table II provides an overview and references prior publication): The first part investigates the scalable visual comparison of code couplings; two novel visualization approaches are proposed for interactively exploring and comparing code couplings (Section II-A). Second, the congruence of coupling structures and modularizations is analyzed in an empirical study in order to investigate how modularity principles are used in practice (Section II-B). Third, it is tried to leverage the multi-dimensionality of coupling for shaping software components that can be extracted for future, independent development (Section II-C). As a starting point, however, a feasibility study, which is not reported in further detail here, was conducted where two concepts of coupling were contrasted and successfully combined in the application of automatically modularizing software systems.

### A. Visual Comparison of Code Couplings

Making code coupling information explorable through visualization is challenging because software systems are complex and easily contain hundreds of entities and thousands of couplings. The representation and comparison of multiple concepts of coupling further complicates the task. In particular, the combination of both challenges—scalability and multi-dimensionality—had not been studied sufficiently. To tackle this combination of challenges, the thesis presents two novel visualization approaches that focus on different aspects of the problem and are based on two different graph visualization paradigms.

1) **Node-Link Approach**: The first visualization technique works with on juxtaposed node-link diagrams, however, varies significantly from traditional node-link approaches by applying a special layout: nodes are arranged on linear vertical axes and each node is split into two ports assembling outgoing and incoming edges. This unusual layout targets at fitting scalable node-link diagrams into small stripes that can be arranged side-by-side enabling graph comparison. An icicle plot is attached reflecting the hierarchical structure of the data. Figure 1 (left) provides an impression of this visualization contrasting three direct structural concepts of coupling (SD.Inh, SD.Aggr, SD.Use) for the JFtp project.

The implemented visualization tool allows manipulating the visualization to retrieve details and to ease the visual comparison process. Basic interactions like focusing and highlighting enlarge parts of the visualization or visually distinguish them from others. While the juxtaposed subdiagrams already support a visual comparison of concepts of coupling, interaction provides the possibility to further enhance the comparison abilities; the tool allows for moving subdiagrams by drag-and-drop; additionally, an advanced highlighting technique of entities enables the comparison of graph features and the possibility to merge concepts of coupling by set union, intersection, and difference helps finding common structures or outliers. Some design alternatives are explored for the approach, among them, different edge bundling strategies [13], [14] and edge splatting [6]. Typical visual patterns are discussed, which help interpreting the visualization. A case study suggests how the approach can be used in different practical software engineering scenarios.

2) **Matrix-Based Approach**: The modularization of a software system is visualized along with code couplings in the presented node-link approach. But there are scenarios where more than one modularization of a system exist: alternative modularizations could be different versions of the system, modularizations created by applying other modularization criteria, or modularizations generated by clustering. Though approaches for visually comparing multiple hierarchies such as modularizations are available [13], [15], these approaches are not suitable for concurrently visualizing code couplings. Due to the need for comparing hierarchies, this approach is visually very different to the previously presented node-link visualization technique; it is based on an adjacency matrix representation of the coupling graphs instead. The matrix is augmented with two different hierarchies, attached to the sides of the matrix as icicle plots. Grayscale rectangles in the background of the matrix encode the similarity of modules reflecting the hierarchical structure of the data. Figure 1 (right), again visualizing the JFtp project: the package structure on the left is contrasted to a clustered modularization at the top; further, two concepts of coupling are visualized within the matrix using different colors.

Like the node-link approach, the introduced matrix approach is implemented as an interactive visualization tool. Besides basic interactions like zooming and selecting, the level of detail for the two modularizations can be adapted interactively: by expanding and collapsing modules in the two hierarchies matching parts of the hierarchy can be retrieved. While it might be acceptable to manually find such similarities small projects such as JFtp, the task becomes tedious for larger projects. Hence, an automatic algorithm is proposed that automatically finds matching levels of detail. To further increase the readability of the visualization, an ordering al-

---

**Table I. Considered concepts of coupling.**

<table>
<thead>
<tr>
<th>Group</th>
<th>Subgroup</th>
<th>Concepts of Coupling</th>
</tr>
</thead>
<tbody>
<tr>
<td>structure</td>
<td>structural dependencies</td>
<td>SD.Inh, SD.Aggr, SD.Use</td>
</tr>
<tr>
<td></td>
<td>fan-out (external)</td>
<td>FO.Inh, FO.Aggr, FO.UseE</td>
</tr>
<tr>
<td></td>
<td>fan-out (internal)</td>
<td>FO.Inh, FO.Aggr, FO.UseI</td>
</tr>
<tr>
<td>history</td>
<td>evolutionary coupling</td>
<td>EC.Sup, EC.Conf</td>
</tr>
<tr>
<td>semantics</td>
<td>code ownership</td>
<td>CC.I, CC.II</td>
</tr>
<tr>
<td></td>
<td>semantic similarity</td>
<td>SS.Tfidf, SS.LSI</td>
</tr>
</tbody>
</table>

**Table II. Outline of the thesis and prior publications.**

<table>
<thead>
<tr>
<th>Part</th>
<th>Objective</th>
<th>Publication</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Feasibility Study)</td>
<td></td>
<td>[5], [4]</td>
</tr>
<tr>
<td>A. Visual Comparison of Code Couplings</td>
<td>Visualize different concepts of code coupling in modular software systems.</td>
<td>[5], [6]</td>
</tr>
<tr>
<td>1) Node-Link Approach</td>
<td></td>
<td>[6], [7]</td>
</tr>
<tr>
<td>2) Matrix Approach</td>
<td></td>
<td>[7], [8]</td>
</tr>
<tr>
<td>3) Evaluation and Comparison</td>
<td></td>
<td>[9]</td>
</tr>
<tr>
<td>B. Empirically Analyzing the Congruence</td>
<td>Analyze the congruence between concepts of coupling and modularity.</td>
<td>[10]*</td>
</tr>
<tr>
<td>C. Component Extraction</td>
<td>Leverage the multi-dimensionality of code coupling for modularizing software systems.</td>
<td>[11], [12]</td>
</tr>
</tbody>
</table>

*a awarded with the Joseph A. Schumpeter-Preis of the Fachbereich IV, University of Trier, 2012*
Algorithm of modules and code entities is introduced that sorts equivalent entities and similar modules to the diagonal of the matrix. Level-of-detail and sorting algorithms were evaluated with respect to layout and interaction improvements based on metrics and examples. A case study provides first evidence on the usefulness of the approach for exploring and restructuring software systems.

3) Evaluation and Comparison: Additional to the case studies, a user study was conducted for exploring how software developers compare different concepts of coupling with the help of visualizations. In particular, the node-link approach presented in this work was contrasted to a matrix approach by Abuthawabeh and Zeckzer [16], which is similar to the matrix approach introduced above—directly comparing the two visualization approaches of the thesis was not possible due to their different focus and abilities. With the help of the visualizations, eight developers analyzed software projects in a realistic scenario based on an explorative study design. A largely qualitative analysis of the results showed that both visualizations can be used for the same program comprehension tasks such as finding key entities, understanding a package, or identifying high-level couplings. Visual comparison of concepts code couplings was integral part of these tasks.

B. Empirically Analyzing the Congruence

In their general theory on modularity, Baldwin and Clark [17] argue that complex systems, such as software systems, need to be modularized to make their complexity manageable. In research literature, multiple desirable attributes are connected to a good modularization of a software system, for instance, comprehensibility [17], [18], abstraction [17], changeability [19], [18], independent development [17], [18], reusability [19], or flexibility [17]. In order to modularize a system, software developers group those code entities that are similar, connected, dependent, related, or in other words coupled by any concept of coupling. Hence, there is a multitude of criteria for modularization: for instance, those code entities are placed in the same module that structurally depend on each other (low coupling and high cohesion) [20], that are connected to the same design decisions (information hiding) [18], or that are related by the communication structure of the development team (Conway’s law) [21].

These modularity principles, more or less, are connected to couplings between code entities. Abstracting from the concrete principles, in each case, pairs of coupled entities should be placed into the same module while pairs of non-coupled entities can be spread across different modules. This implies that, in a good modularization, there exists certain congruence between code couplings and modularity: couplings are assumed to mostly connect entities in the same module rather than in different modules. Different modularity principles can be mapped to different concepts of code coupling as summarized in Table III. For instance, the principle of information hiding assumes the locality of co-changes, which is reflected in a high congruence between evolutionary couplings and the existing modularization of the system.

Although the conjectured relationships are different and are founded on diverse rationale, they share a specific characteristic: assumed that the relationships are correct, the particular modularity principle is met if the code couplings of the respective concept are largely local with respect to modularity. Local, in this context, means that the couplings mainly connect code entities contained in the same or similar modules instead of entities from very different modules. The coupling-modularity congruence is introduced as a measure for empirically in-
While perfect segregation is the idealistic but unreachable goal, the modularization of components requires that components are decoupled. This is closely related to code couplings—future independent development of the extracted component. Similar as software clustering, component extraction is enabling future independent development of the extracted component. An automatic extraction of a single component instead of modularization approach is proposed that focuses on the semi-interactive [25], [26]. Considering these issues, a novel algorithm is tested with multiple concepts of coupling similar to the empirical study on modularity (Section II-B). The observed results are largely consistent to the results reported for the congruence between the concepts of coupling and modularity: $SD.Inh$ and $FO.InhI$ perform best, followed by other concepts of direct structural coupling and evolutionary coupling. Surprisingly, however, the merging of concepts does not improve the results of the approach reasonably, but acts more as a way to average the outcome.

Interacting with the component extraction result requires an intuitive and scalable user interface that represents the proposed decomposition and relevant couplings. Two versions were developed: the first prototype was evaluated in small user projects; based on the experience from this study, an improved interface was designed as an IDE plug-in (Figure 3). The two user interfaces mainly differ in the approach used for representing couplings between the components: while the prototype employed explicit visual links between entities, the IDE plug-in works with aggregated lists of couplings to enhance scalability. Both interfaces share a split main view with three columns—the original system on the left, the extracted component on the right, and the contract in between.

Finally, it is evaluated what concepts of coupling are most suitable for component extraction and how they might be combined to further improve the results. The component extraction algorithm is tested with multiple concepts of coupling similar to the empirical study on modularity (Section II-B). The results hint at a particular impact of the principle of low coupling and high cohesion and of information hiding (Table III, Congruence).

### C. Component Extraction

Code couplings are used for automatically clustering software systems [24]. Many software clustering approaches, however, share the problems that they ignore existing modularization, change every part of the system ruthlessly, and are not interactive [25], [26]. Considering these issues, a novel modularization approach is proposed that focuses on the semi-automatic extraction of a single component instead of modularizing the whole project fully automatically. The target is enabling future independent development of the extracted component. Similar as software clustering, component extraction is closely related to code couplings—future independent development of components requires that components are decoupled. While perfect segregation is the idealistic but unreachable goal, component extraction should at least try to minimize coupling. A related technique is component extraction refactoring [27].

As a starting point, it is assumed that the developers already have some rough idea which part of the software they want to extract: they identify two sets of key entities (i.e., classes and interfaces), one providing seeds for the extracted component, the other specifying the parts of the program that should not be extracted. A component extraction algorithm computes a minimal cut in the coupling graph between the two sets. The result is the basis for proposing a component to extract and for creating a so-called contract, which acts as an interface between the component and the remaining system. Being an interactive approach, users may modify the set of key entities based on the proposed result and rerun the algorithm.

Interacting with the component extraction result requires an intuitive and scalable user interface that represents the proposed decomposition and relevant couplings. Two versions were developed: the first prototype was evaluated in small user projects; based on the experience from this study, an improved interface was designed as an IDE plug-in (Figure 3). The two user interfaces mainly differ in the approach used for representing couplings between the components: while the prototype employed explicit visual links between entities, the IDE plug-in works with aggregated lists of couplings to enhance scalability. Both interfaces share a split main view with three columns—the original system on the left, the extracted component on the right, and the contract in between.

Finally, it is evaluated what concepts of coupling are most suitable for component extraction and how they might be combined to further improve the results. The component extraction algorithm is tested with multiple concepts of coupling similar to the empirical study on modularity (Section II-B). The observed results are largely consistent to the results reported for the congruence between the concepts of coupling and modularity: $SD.Inh$ and $FO.InhI$ perform best, followed by other concepts of direct structural coupling and evolutionary coupling. Surprisingly, however, the merging of concepts does not improve the results of the approach reasonably, but acts more as a way to average the outcome.

### III. Future Research Challenges

The described work considers code coupling as a multi-dimensional construct. This shift from one-dimensional to multi-dimensional couplings opens up a wealth of questions—only a few of them have been answered. A particular focus has been on how multi-dimensional couplings interact with modularity. However, other applications than modularization have stayed untouched, such as change impact analysis or bug detection where code couplings seem to have a similarly important role. Understanding couplings as multi-dimensional will probably also have an impact on these domains. While this is a general observation, more specific research challenges are the following.

**RC1: The Meaning of Coupling**—How to define coupling (or a related term) so that the definition matches the intuition of developers, is measurable, and reflects multi-dimensionality?

The definition by Fowler [1] used in this work is based on potential future co-changes (Section I). It is open and takes the
perspectives of developers but is difficult to map to a metric. Considering multiple concepts of couplings is possible but the selection of concepts lacks empirical evidence. In a recent study, Bavota et al. [28] partly address the above research challenge: they investigated which concept of coupling is most related to the intuition of developers. Pairs of code entities, which are (not) coupled with respect to a certain concept, were presented to developers, who rated the strength of coupling. The results suggest that, in particular, semantic similarity and structural dependencies match well the developers’ idea of coupling. While this is an interesting first step, more research need to be conducted in this direction.

**RC2: Combination of Code Couplings**—How to combine multi-dimensional couplings to a single concept of coupling?

A related question to finding a consistent and measurable definition of coupling is how to combine several concepts. While dividing couplings into several concept might be appropriate in a first step, most algorithmic approaches processing couplings require a single concept. Since only using one of the concepts means throwing away much information, it is desirable to combine different data sources. In this work, simple approaches were tested based on the union of couplings: while this was successful in the feasibility study for the application of software clustering with two concepts of coupling, a similar approach for component extraction combining more concepts of coupling showed only limited success. In general, a multitude of ways exist how to combine several concepts of coupling [4]; a systematic evaluation which works best with respect to different applications has not yet been conducted.

**RC3: Interactive Software Modularization**—How to design a general recommendation tool for modularizing software systems based on multidimensional code couplings?

Component extraction was only a first try for leveraging multi-dimensional couplings in practice. It focused on a special use case of modularizing a software system; combining concepts of coupling was only partly successful. An open research question still is how the advantages of component extraction—in particular the interactive and transparent approach—can be transferred to general software clustering. While some works already propose solutions into this direction [29], [30], using multi-dimensional couplings in such an approach is still a challenge. Multiple concepts of coupling might help better explaining developers proposed remodularizations.

**IV. LESSONS LEARNED**

Writing a doctoral dissertation, no matter how fascinating, is a long and exhausting process. In this section, I report some lessons learned divided into issues related to software engineering research (SE) and general observations on writing a doctoral thesis (DT).

**Lesson SE1:** Changing a software system has to be controlled by the developer.

Coming up with a good solution of a software engineering problem is often not enough: developers will not apply the proposed changes if they do not understand why they should and are not in control of the process. Taking software clustering as an example, dozens of approaches exist, but they seem to be rarely applied by developers because they work automatically [25]. While I first also experimented with software clustering in the feasibility study (see Table II), by now, I believe that only user-controlled, semi-automatic approaches like component extraction (Section II-C) or the interactive modularization approach outlined as future work (RC3) have the chance to be accepted by developers.

**Lesson SE2:** There is not much fundamental research and theory on software engineering.

Software engineering research seems to be very practical: hardly any paper does not present a specific application of the results. Searching for literature on modularity principles, I found mostly works from the 1970s (Section II-B), but relatively few follow-up papers with a comparable focus on theory and fundamentals. Only recently, people within the field start to rediscover this kind of research (e.g., [10], [28]).

**Lesson DT1:** Starting to write early helps keeping focused.

Just to take some notes on related work, I started to compile a document for the thesis early (in the first year). In retrospective, I have been lucky: working regularly with the document to extend my notes, I unwillingly thought about the structure of the thesis, the story I wanted to tell, and what work currently was most urgent. This kept me from losing focus during the four years of my dissertation.

**Lesson DT2:** Scale introduces new complexity.

When I began compiling the main parts of the thesis, through publishing papers, I already had some experience...
on writing scientific texts. But I learned that it is not just more work to write a longer text such as a thesis, but it introduces new complexity; it is much harder to use consistent terminology and telling a story is getting multifaceted.

V. Conclusion

The summarized thesis analyzes multi-dimensional code couplings in software projects, i.e., different concepts of code coupling derived from multiple data sources. The goal was taking the next step in understanding these couplings by providing means for exploration, by presenting empirical findings, and by proposing approaches for improving the modularization of software projects. While the results are firstly limited to the studied Java systems and cannot be directly generalized to other systems, this work also provides analysis tools (visualizations and software development tools) as well as methods (metrics, evaluation methods, and algorithms); these can be used for analyzing other datasets and for replicating the results. Distinguishing features of the work are the large set of concepts considered for code coupling and the focus on fundamentals instead of concrete applications scenarios.

The thesis considers code coupling as a multi-dimensional construct rather than a one-dimensional one. While, occasionally, a small set of concepts of coupling have been compared or combined in a particular application, this work is one of the first that systematically investigate the idea of multi-dimensional code couplings. For using the full potential of multi-dimensional code couplings, an open research question, however, still is how to best combine the different concepts. In general, the work can be considered as a first step towards the ambitious goal of understanding multi-dimensional code couplings or—to phrase it in the words of Fowler—how changing one module in a program requires changing another module.

REFERENCES