Identifying Modularization Patterns by Visual Comparison of Multiple Hierarchies

Fabian Beck, Jan Melcher, and Daniel Weiskopf
VISUS, University of Stuttgart, Germany
Email: {firstname.lastname}@visus.uni-stuttgart.de

Abstract—**Software is modularized to make its high complexity manageable. However, a multitude of modularization criteria exists and is applied. Hence, to extend, reuse, or restructure a system, it is important for developers to understand which criteria have been used. To this end, we provide an interactive visualization approach that compares the current modularization of a system to several software clustering results. The visualization is based on juxtaposed icicle plot representations of the hierarchical modularizations, encoding similarity by color. A detailed comparison is facilitated by an advanced selection concept. Coupling graphs, which form the basis for software clustering, can be explored on demand in matrix representations. We discuss typical modularization patterns that indicate criteria used for structuring the software or suggest opportunities for partial remodularization of the system. We apply the approach to analyze 16 open source Java projects. The results show that identifying those modularization patterns provides valuable insights and can be done efficiently.**

I. Introduction

Like signposts at a street, the modularization of a software system acts as a guide to understand and navigate a software system. However, the information provided is often quite sparse: modules carry a name and are subdivided into submodules. Information on what was the criterion to create the module is often missing. Different strategies might have been applied for different parts of the system—metaphorically speaking, a modularization is a patchy mix of signposts, sometimes rather misleading than guiding.

Software clustering approaches automatically derive a consistent modularization based on predefined criteria. However, software clustering techniques share a number of unsolved issues: First, replacing a historically grown modularization by an artificially created one destroys the mental map of the system that the developers have developed over time. Second, it is hard to automatically come up with meaningful names for automatically generated modules. Finally, there are several criteria to structure a software system—the clustering algorithm usually follows a single criterion (or a mix of few). The process always recommends only one solution.

Our approach breaks with this one-fits-all paradigm of software clustering. It accepts that there exist competing valid modularizations of software systems that developers mix into a single software modularization applying different criteria. We also acknowledge the efforts that developers have already invested into structuring the system—the current modularization is the primary artifact visualized by our approach. We provide support for understanding the criteria used to modularize the system. Our visualization compares the current modularization to different computationally derived modularizations.

With this approach, we aim to support both software developers and software engineering researchers:

- **Use Case I (Developer):** To extend, reuse, or remodularize parts of the system, a developer needs to identify and understand the individual criteria that were applied to construct a specific module.
- **Use Case II (Researcher):** To study how modularization criteria are applied in practice, software engineering researchers want to study typical patterns of modularization across different projects.

While our approach is generic enough to work with any hierarchical structure of software systems, we focus on Java systems as a specific example. In Java, classes and interfaces are organized into a hierarchical package structure.

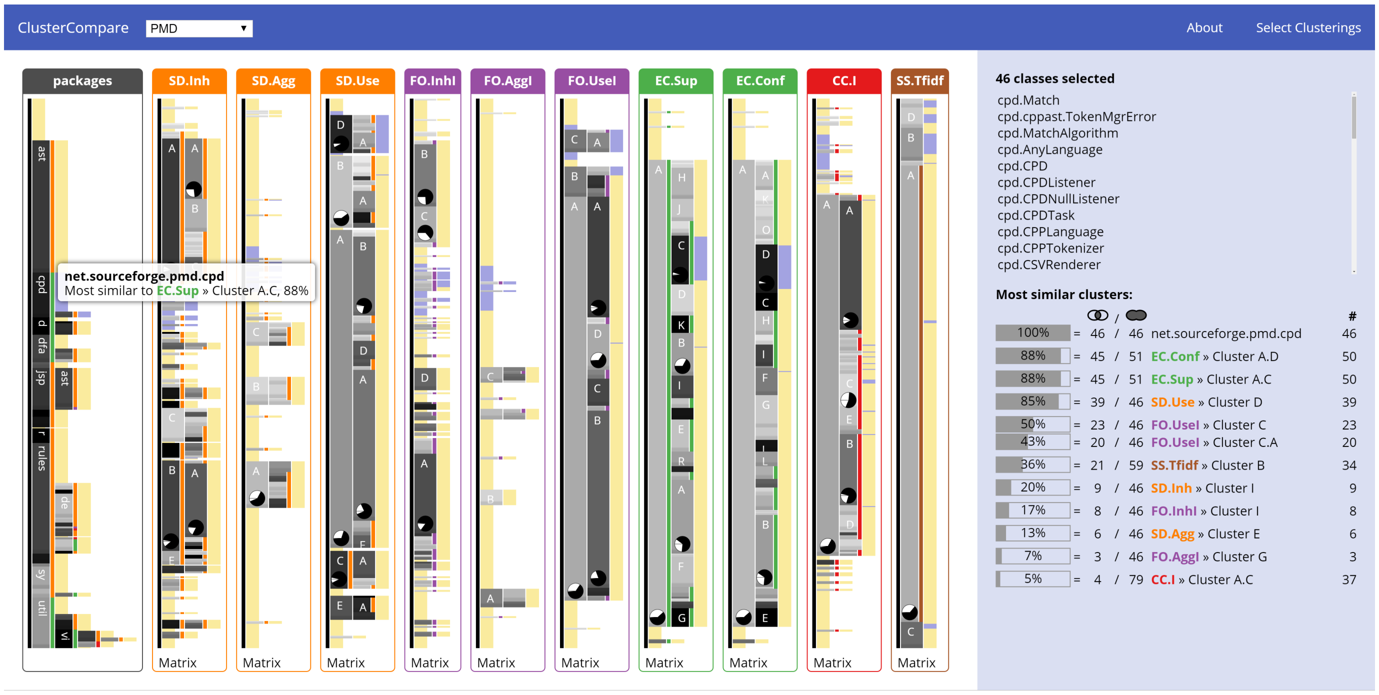
In our interactive visualization (Figure 1), we represent the package structure and clustered modularizations as hierarchies, where the package structure is always placed to the left. We use space-efficient *icicle plots* [1] as hierarchy representations, which employ subdivided and stacked boxes to represent hierarchical containment. The color coding of the hierarchy elements indicates similarities of packages to any of the modules derived by clustering, and vice versa. An advanced selection concept allows for investigating those similarities in detail—a sidebar on the right provides further information on the current selection. We implemented our approach in a Web-based system called *ClusterCompare*; a demo of the system including the analyzed data is available online.[1]

In this work, we first give an overview of software clustering and visual support to compare hierarchies (Section II). We then explain the visual encoding and interactions that comprise our visual comparison approach of software modularizations (Section III). This visual analysis approach can be leveraged for identifying typical modularization patterns (Section IV). Finally, we present results gained by using the approach for studying 16 open source software projects (Section V) and arrive at our conclusions (Section VII).

II. Related Work

Software clustering is a well-studied area of software engineering research. Traditionally, structural dependencies like method calls have been used to automatically obtain a

[1] https://clustercompare.jan-melcher.de/



net.sourceforge.pmd.cpd
Most similar to EC.Sup » Cluster A.C, 88%

46 classes selected

cpd.Match
cpd.cppast.TokenMgrError
cpd.MatchAlgorithm
cpd.AnyLanguage
cpd.CPD
cpd.CPDListener
cpd.CPDNullListener
cpd.CPDTask
cpd.CPPLanguage
cpd.CPPTokenizer
cpd.CSVRenderer

Most similar clusters:

			/			#
100%	=	46	/	46	net.sourceforge.pmd.cpd	46
88%	=	45	/	51	EC.Conf » Cluster A.D	50
88%	=	45	/	51	EC.Sup » Cluster A.C	50
85%	=	39	/	46	SD.Use » Cluster D	39
50%	=	23	/	46	FO.Usel » Cluster C	23
43%	=	20	/	46	FO.Usel » Cluster C.A	20
36%	=	21	/	59	SS.Tfidf » Cluster B	34
20%	=	9	/	46	SD.Inh » Cluster I	9
17%	=	8	/	46	FO.Inhl » Cluster I	8
13%	=	6	/	46	SD.Agg » Cluster E	6
7%	=	3	/	46	FO.Aggl » Cluster G	3
5%	=	4	/	79	CC.I » Cluster A.C	37

Fig. 1. Interactive visualization showing project *PMD* with juxtaposed hierarchy representations in the left panel (first plot: package structure; following plots: different clustering results) and details on the current selection in the right panel (Package `ast` selected).

modularization of the software system using clustering algorithms [2], [3], [4], [5], [6]. However, code entities are not only grouped due to their structural embedding, but the design of modules might incorporate domain-specific knowledge, code similarity, or historic creation and changes of the source code. Hence, employing other code coupling definitions produces meaningful clustering results as well, such as using semantic similarity of the vocabulary used in the code (*semantic similarity*) [7] or co-changes of code entities (*evolutionary coupling*) [8], [9], [10]. Combinations of these couplings often produce better results [3], [4], [11]. In general, different concepts of coupling play a role in practice to modularize systems [12] and the coupling developers perceive is not limited to structural dependencies [13].

It is appropriate to derive a modularization from scratch for legacy systems without any useful modularization. However, in most cases, there already exists a meaningful modularization of a software system, which is often not perfect and might require updates. Müller et al. [14] recommend not to fully automate the process and Glorie et al. [15] and Rama [16] warn not to ignore the already existing structures. There are some approaches that address the issues by only partly changing the system, for instance, by extracting only specific components [17], [18], [19] or using the current modularization as a starting point for optimization [20].

Visualization could help provide an explorable comparison between the current modularization of a system and any clustering result. For instance, two juxtaposed visualizations connected by links show similarities between the modularization and the clustering [21], [22]. Alternative approaches for encoding similarities use matrices [23] or nesting the clustering result into the representation of the current modularization [24]. Other related approaches visualize the evolution of a modularization as a timeline of juxtaposed and connected hierarchies [25]. Vehlow et al. [26] extend this approach to showing dependency matrices in each timestep and comparing different dynamic hierarchy sequences.

There exist more work on visual hierarchy comparison in other domains, some of approaches cover the comparison of more than two hierarchies applying coloring for similarity encoding [27]. In context of phylogenetic trees, Bremm et al. [28] juxtapose node-link representations of hierarchies. Different measures show global and local similarities. When selecting a reference hierarchy, the approach is similar to ours. However, they focus on comparing more, but smaller hierarchies. For fewer but larger hierarchies, Graham and Kennedy [29] work with juxtaposed icicle-plot-like representations, but use a different coloring approach and no reference hierarchy. *TreeDyn* [30] provides, among other views, a juxtaposed representation for biological trees, but focusing on annotation features. For an extensive survey of visual hierarchy comparison, we refer to Graham and Kennedy [27].

Our approach targets at the comparison of the current modularization to a set of hierarchical clustering results. It is different from previous work because it is the first multi-hierarchy comparison approach tailored for a software clustering scenario. It does not combine multiple data sources into one clustering result, but presents the results of alternative clustering runs. The visualization extends state-of-the-art visual hierarchy comparison approaches for this scenario.

III. Visual Comparison Approach

As shown in Figure 1, we visualize different modularizations of software systems and offer visual cues and interactions to compare them. We assume an object-oriented programming style and use classes and interfaces as atomic pieces of software. Please note that, in the following, the term *classes* refers to both *classes* and *interfaces* for simplification. These atomic entities are organized in a hierarchical package structure. This current package structure of the system is used as reference modularization for comparison to generated modularizations. Because of this asymmetry in roles—current, or *primary*, hierarchy and generated *secondary* hierarchies—, we use an asymmetric visualization approach. Small multiples [31] of icicle plots [1] represent the hierarchies in a space-efficient visualization. Similarities between clusters and packages are color-coded in the nodes (rectangular blocks) of these plots. Interactions allow users to explore the dataset. This section introduces the general approach together with important design decisions in detail; a tutorial for users is available online.[2]

A. Small Multiples of Icicle Plots

Icicle plots [1] are a space-efficient hierarchy visualization approach because neither links (like in traditional node-link representations) nor indentation (like in indented hierarchy diagrams often used in file browsers) require additional space. Nodes at lower levels are larger and thus have room to display their label, whereas leaf nodes are compressed to a thin slice and do not provide enough space for labels. To clearly distinguish class nodes from package and cluster nodes, we draw class nodes smaller than inner nodes and use a yellow background color for them. The topmost levels of the package hierarchy are usually identical for all classes and get aggregated into a single, thin root node. Horizontally, the rectangles in the icicle plot are separated by a small margin. Vertically, this approach would cause visual clutter because some of the nodes are very small in height. Instead, each node has a small color gradient at the bottom that distinguishes it from the adjacent node.

The visualization compares several secondary hierarchies to one primary hierarchy. We use a small multiples visualization [31] to give an overview of multiple hierarchies at a time. While is possible to support the comparison by links connecting the hierarchies [21], [22], this does not scale to a comparison of multiple hierarchies: We could line up the icicle plots and draw links between adjacent hierarchies, but since all hierarchies are to be compared to the package hierarchy, this would not meet our requirements. Therefore, we decided to display the icicle plots side-by-side without links. We assign a color to each clustered modularization for identification and apply it to the surrounding borderline; we use the same color to group related clustering results.

B. Ordering of Nodes

The package structure as primary hierarchy is ordered alphabetically to resemble the view in an IDE or file explorer and to simplify finding a specific class or package. Since inner nodes of the secondary, clustered hierarchies only have artificial labels, their order is not predefined. We sort the nodes of secondary hierarchies in similar orders as in the primary hierarchy. This simplifies finding the same class in multiple icicle plots. A second advantage is that a continuous list of nodes in one hierarchy is likely to be continuous in other hierarchies as well.

To achieve a consistent ordering, we use a sorting algorithm similar to the one introduced by Holten and van Wijk [21], which is based on the work of Sugiyama et al. [32]. They also use icicle plots, but connect corresponding nodes with lines; the algorithm sorts both hierarchies to reduce edge crossings. Our problem is similar: while our nodes are not visually connected, reduction in edge crossings also results in reduction of distance between connected nodes.

Since the order of the primary hierarchy is fixed, we can use a simplified sort algorithm that just requires one traversal. The general idea is to order nodes by the position of the barycenter of their corresponding nodes in the other hierarchy. Inner nodes are represented by the set of contained leaf nodes. For instance, if the classes in a cluster are on average near the bottom of the primary hierarchy, this cluster has a tendency to be moved to a low position. The algorithm recursively traverses the hierarchy and reorders the hierarchy.

C. Similarity Encoding

To simplify the comparison of hierarchies further, nodes of the secondary hierarchies are color-coded to show their similarity to nodes of the primary hierarchy. The gray value of a node represents the similarity of it to the best-fitting package in the primary hierarchy. Vice versa, to find package nodes that have a cluster with high similarity, the same gray value is used for package nodes. Instead of selecting the best node from only one secondary hierarchy, the most similar node of *all* secondary hierarchies is used. Figure 2 illustrates how the similarity values of nodes are calculated.

Formally, the package set P as well as each of the clustered modularizations M_i is a family of sets of classes C, defined as $P = \{p_1, \dots, p_n\}$ and $M_i = \{m_{i,1}, \dots, m_{i,l_i}\}$ ($M = \cup_{i=1}^{k} M_i$). Each package p or clustered module m is a set of classes from C; the hierarchical structure of packages and clustered modules is only implicitly encoded by subset relations. Further, $\mathrm{sim}(p, m)$ denotes a similarity metric between a package p and a clustered module m. To

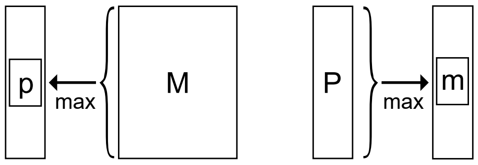


Fig. 2. Illustration of how similarity values are aggregated according to Equation 1a (left) and Equation 1b (right).

[2]https://clustercompare.jan-melcher.de/about/

assign each package and clustered module a similarity value used for coloring, we define:

$$\mathrm{sim}(p) = \max_{m \in M} \mathrm{sim}(p,m) \tag{1a}$$

$$\mathrm{sim}(m) = \max_{p \in P} \mathrm{sim}(p,m) \tag{1b}$$

This similarity value is mapped to a gray scale from light gray (low similarity) to black (high similarity). White is not part of the color scale because otherwise the nodes would not be visible against the background of the visualization. We use the Jaccard coefficient as a default similarity metric often applied in hierarchy comparison [22], [23], [24], [28], [33]. The Jaccard coefficient is the number of classes that are leaves of both nodes divided by the number of classes that are leaves of either node:

$$\mathrm{sim}(p,m) = \frac{|p \cap m|}{|p \cup m|} \tag{2}$$

Hence, for calculating the similarity metric of two nodes, we take only their leaf nodes into account. Clusters and packages that contain the same classes are considered maximally similar, independent of how those classes are substructured. Since the Jaccard coefficient is a ratio between 0 and 1, we list all similarities as percentage values in the user interface.

The background color shows *how* similar a node is to other nodes, but we also want to determine *which* node is the most similar one. Nodes are decorated with colored vertical bars. The color of this bar in the primary hierarchy corresponds to the clustering that contains the most similar cluster. For every package, the most similar cluster is also highlighted with the same bar in the respective secondary hierarchy. Once the user is familiar with the color coding, this color bar provides a quick way to identify which coupling concepts are dominant in the package structure (e.g., orange colored *structural dependencies* in Figure 1).

In inner nodes of the secondary hierarchies, pie charts further indicate how the nodes relate to the most similar package. They are separated vertically: the top part shows how many classes in the cluster are not contained in the package, and are thus *additions*; the bottom half relates to the number of classes missing in the cluster, called *removals*. Both values are presented relative to the total number of classes in the package and in the cluster. The background color of the pie chart is black and the filled parts are colored in white. Thus, on clusters with 100% similarity, the pie chart becomes invisible. This is an intentional effect because there are neither additions nor removals that need to be noted.

When the mouse hovers a node, a tooltip is shown, which contains the full label of a class, package, or cluster (Figure 1). For inner nodes, the label of the most similar cluster and its clustering is given, along with the similarity value. Also, a larger version of the pie chart is displayed for all clustered modules. This is especially important for nodes that are too small for containing a pie chart.

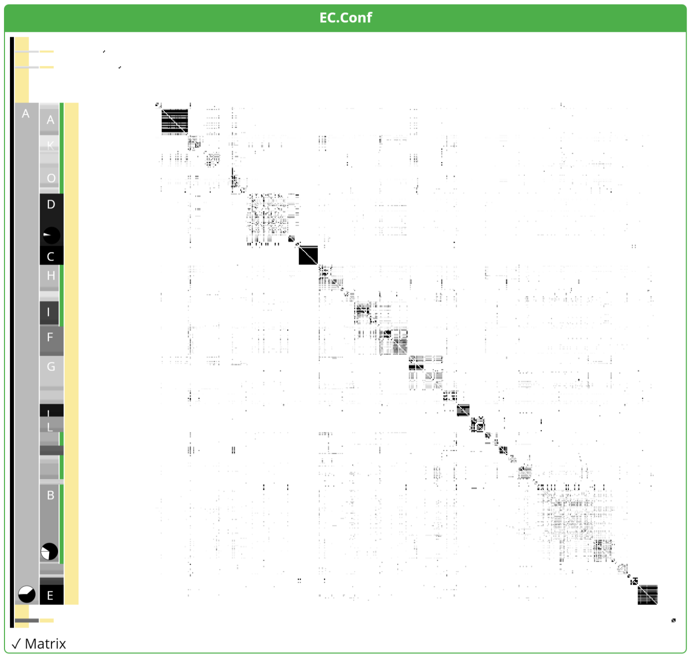


Fig. 3. Adjacency matrix visualization showing the evolutionary coupling graph EC.Conf and the related clustering result for project *PMD*.

D. Coupling Matrix

While the icicle plots visualize the clustering results, they are already an abstraction and might hide important details. For deeper inspection, it is helpful to see the raw coupling information that was used to create the clustered hierarchy. Therefore, the visualization provides an adjacency matrix representation for all icicle plots that shows the underlying coupling graph. Matrices can be toggled via a button below the icicle plots and are shown directly beside them (Figure 3). The icicle plot works as a legend for the matrix: rows of the matrix align with the leaf nodes of the icicle plots; columns have the same order. Such hierarchical adjacency matrix representations, also called *dependency structure matrices*, are a common way to visualize hierarchically structured coupling graphs [23], [34], [35]. We decided against adding links to connect the nodes because such an approach does not visually scale well [36]. A scalable alternative would have been to apply *parallel edge splatting* [37], which makes the graph artificially bipartite and uses straight (splatted) links. This approach, however, is likely more unfamiliar to software engineers.

E. Interactive Exploration

One basic but powerful feature for small multiples visualization is *brushing and linking* (i.e., connecting the different visualizations by linked selections of visual elements). It allows the user to trace one item across all plots. Our visualization employs brushing and linking on class level, i.e., it is possible to select a set of classes that will then be highlighted across all icicle plots.

We use modifier keys to provide a flexible selection mechanism without requiring different input modes. The user can simply click on any node to select this node. If the clicked node is an inner node, all leaf nodes are selected. Any former selections are discarded. If the *Ctrl* key is pressed while clicking, the new selection is added to the previous one. This allows us to select multiple disjoint nodes. It is also possible to remove nodes from the selection by *Ctrl*-click and also works for inner nodes if all their leaf nodes are selected. This mechanism is similar to how file explorers and other widely known applications work, and therefore familiar to the user.

Besides the main selection, nodes can also be highlighted by just hovering them with the mouse. This will highlight the leaves of the hovered nodes as well as the corresponding leaf nodes in the other hierarchies with a different color. We have chosen a blue color for the main selection, as it is known from many existing applications. The hover selection will just darken nodes—if they are contained in the main selection, the color is darker blue, otherwise darker yellow. This allows us to combine both selections.

Main selection does not only highlight nodes but also provides additional information in the sidebar. At the top, the selected classes are listed. Clicking on a class will show the source code of the class. Below, there is a list of clusters and packages that are most similar to the selection. Similarity is measured as described in Section III-C. Of each clustering, there is at least one cluster shown. If there are multiple clusters with similarity values above a certain threshold (33% in our case), those second- and third-best clusters are shown directly below the best cluster in their clustering. Thus, the clusters are grouped by their origin.

The clusters are presented in the form of a table. The leftmost column is a visual indicator for the similarity. The percentage is shown inside a rectangle that is filled according to the similarity value. In the second and third column, this value is explained in terms of Equation 2, i.e., the sizes of the intersection and union of the cluster and the selection are given. Next, the cluster label is displayed and prefixed by a colored clustering identifier. In case of packages, the package name is shown. The last column contains the number of classes of each cluster. Hovering over an item in the list of similar clusters highlights the contained classes in the hover selection mode. This allows users to quickly compare the selection to the cluster. The icicle plot of the hovered clustering is also highlighted with a shadow. Clicking a cluster selects it using the main selection mode.

To go back to an earlier selection, an undo/redo feature is provided through the backward and forward buttons of the browser. The selection is encoded in the URL to allow users to bookmark or share a particular selection.

Users choose which clusterings are shown in an additional pane that can be opened in the sidebar. It provides a short explanation for each clustering result. Once selected, the icicle plots can be rearranged via drag and drop.

IV. Software Modularization

Through the use of small multiples, color coding, and interactive selections, our approach supports the comparison of multiple clustering results to a package structure. While different clustered modularizations might be produced by different clustering algorithms, we want to focus here on varying data sources of clustering. For all modularizations, we use the same algorithm, but vary the coupling graph that is used as a basis for clustering.

TABLE I
Considered concepts of code coupling [12].

SD	*Structural dependencies* based on *inheritance* of classes (Inh), *aggregation* of classes through fields (Agg), or other *usage* of classes, e.g., method calls, usage as local parameter, etc. (Use).
FO	*Fan out* coupling, i.e., classes are similar if they share similar dependencies to *external* libraries (InhE, AggE, UseE) or to project *internal* classes (InhI, AggI, UseI).
EC	*Evolutionary coupling* constructed from co-changes of classes based on *support* (Sup) and *confidence* (Conf) metrics [11].
CO	*Code ownership* connecting files that have the similar set of authors based on *binary* author assignments (Bin) or *proportional* (Prop).
CC	*Code clone* coupling indicating coverage of the same *type I* (I) and *type II* (II) clones.
SS	*Semantic similarity* of the code retrieved from shared vocabulary applying a *term frequency–inverse document frequency* measure (Tfidf) and *latent semantic indexing* (LSI).

A. Code Coupling and Modularization

Each of the coupling graphs represents a different modularization criterion that might explain parts of the current modularization or could suggest improvements in the modularization. Beck and Diehl [12] survey different definitions of code coupling used in literature, such as structural dependencies (SD), fan-out coupling (FO), evolutionary coupling (EC), code ownership coupling (CO), code clone coupling (CC), and semantic similarity (SS). Each of these general coupling concepts is subdivided into different instances representing specific coupling metrics. Table I gives an overview of these coupling concepts and provides short explanations; for more information, we refer to the original publication [12]. We apply these definitions in the following for obtaining graph structures that we use for clustering.

These concepts of coupling are connected to different modularization criteria advocated in literature [12]. For instance, the well-known concept of *low coupling and high cohesion* [38] is related to *structural dependencies* (SD): applying the principle should lead to dense clusters of structurally connected classes within packages that have only few connections to other packages. In contrast, *information hiding* [39] is conjectured to be connected to *fan-out coupling* (FO) and *evolutionary coupling* (EC). In this way, insights obtained with our visualization approach hint at higher-level design guidelines that might have been applied by the developers.

B. Modularization Patterns

Our visualization approach now supports us to identify patterns in the visualization that indicate criteria used for modularizing a specific part of the system. In case a package is not (or only partly) matched by any of the modules in the different clustering results, the visualization provides hints how to remodularize the system. While color-coded similarities and pie diagrams are first indicators, usually an interactive exploration helps to fully understand a specific pattern.

In particular, we define a *modularization pattern* as a specific type of similarity between a package and one of the clustered modules observable in our visualization. We discern three main patterns that are further divided into subpatterns. We came up with these patterns following an iterative approach, discussing the patterns among the authors and applying them to two pilot examples, *PMD* and *JHotDraw*, that we introduce in further detail in Section V-B. After identifying the patterns, we checked that they are applicable to other projects by expanding the analysis to 14 other projects (Section V-C).

P1: Match—Dark gray or black boxes in the package structure identify packages that are well-matched by one of the clusters. The colored bar hints at the origin of the cluster, while the specific cluster can be retrieved by hovering over or selecting the package. In general, a *match* is a strong indicator that a certain information was used to modularize the respective package. In some cases, the *match* between a package and a cluster is perfect: a 100% agreement encoded in black (**P1.1: Perfect Match**). In other cases, there is still a small difference between the package and the best matching cluster, which might be that the cluster contains some additional elements (**P1.2: Addition Match**) or fewer (**P1.3: Removal Match**). This information can be easily obtained from the pie chart of the cluster. To explore these added or removed elements one-by-one, for *addition matches*, the user first selects the cluster and then *Ctrl*-clicks the package (vice versa, for *removal matches*). Please note that *addition* and *removal matches* can occur for the same package–cluster match. In addition, the clustering might suggest a subdivision (**P1.4: Subdivision Match**): the matched cluster is further divided into fine-grained clusters in a second level colored in gray. This suggest a finer or alternative substructure of the current package, which might be considered for remodularization.

P2: Partial Match—If only an incomplete but significant part of a package is matched by any of the clustered modules, the package has a medium gray background. Such a *partial match* can be caused by two situations: First, if the matched classes are a subset of the matched cluster, this indicates that other classes have been detected as related by the clustering result. If this clustered module is not yet well-matched by any of the packages, it might make sense to *merge* the classes of the partially matched cluster into a larger package as part of a remodularization (**P2.1: Suggested Merge**). In contrast, if the cluster even better matches another package (typically, the parent package of the partially matched one), the *merge* is already reflected in the package structure, hence is *confirmatory* (**P2.2: Confirmatory Merge**). Second, as a complement, the partial match might suggest splitting a package when there exist several partly matched, independent clusters (**P2.3: Suggested Split**). If this split is already realized in the package structure (here, by well-matched subpackages), it is confirmatory (**P2.4: Confirmatory Split**). Blends of merges and splits are possible, even likely, whenever the partially matched clusters contains neither a perfect subset nor superset of elements. Again, interactive selection and details-on-demand help users discern these subpatterns.

P3: Miss—A light gray package indicates a low agreement to any clustering result—none of the clustering results explains why the package is constructed like this. There might exist a criterion, but extra information is required to identify it.

C. Patterns in Application

As mentioned above, certain subpatterns can appear together for a package. Also, some packages cannot be unambiguously assigned; for instance, there might be corner cases between a *match* and a *partial match*. The patterns are only intended to provide a vocabulary and describe idealized scenarios. We also do not define specific thresholds to discern between the patterns because these numbers might be project-specific and could depend on the context the approach is used in. To clearly focus on the package structure as our primary hierarchy, the degree of similarity of the package—not the cluster—discerns the type of pattern. But if the user wants to analyze a certain clustering, the proposed patterns are also applicable in reverse by starting at the similarity value assigned to a specific cluster and then comparing it to the matched packages of this cluster. In practical application, it is not only important which pattern is assigned to a package, but of course, also to which of the clustering result it is most similar. If the developers agreed to follow a certain guideline to modularize the packages (e.g., *low coupling and high cohesion* [38]), matches to some of the clusterings are intended (e.g., *structural dependencies*). When packages do not match any intended clustering result, the developers might want to refactor them according to the suggested clusterings.

V. Results

We demonstrate the usefulness and practical relevance of the presented approach by applying it to open source Java projects. In particular, we use the dataset collected and described in previous work [12] to investigate the congruence of code coupling and modularity. It comprises 16 small to mid-size projects (78–679 classes, 6–83 packages) of different kind. Each project is modeled as a set of 17 coupling graphs on class level, covering the different concepts of coupling described in Table I. To derive different modularizations, we fed each of the coupling graphs into a hierarchical graph clustering algorithm. In particular, we used *infohiermap* [40], a state-of-the-art clustering approach that automatically computes a meaningfully coarse hierarchical structure by detecting densely connected groups (i.e., modules) of graph nodes (i.e., classes).

In the following, we present the analysis of this data using our visualization approach. First, we demonstrate that a comparison of multiple hierarchies is necessary by showing that indeed different clustering results provide meaningful matches to the current package structure. Second, we investigate two systems in detail and give qualitative examples of meaningful insights gained. Finally, we scale the analysis to all 16 projects and show a quantitative overview of detected patterns.

TABLE II

Percentage of best matching packages of software projects regarding the specific clustering results; gray highlights high values; percentage values might sum up to more than 100% per project because multiple best matching clusters are possible.

		SD			FO						EC		CO		CC		SS	
	#pkg	Inh	Agg	Use	InhE	AggE	UseE	InhI	AggI	UseI	Sup	Conf	Bin	Prop	I	II	Tfidf	LSI
Checkstyle	20	20%	5%	20%	-	-	-	-	-	-	40%	40%	-	-	10%	-	-	5%
Cobertura	15	13%	33%	27%	-	-	-	-	7%	-	7%	-	-	-	20%	-	-	-
CruiseControl	23	17%	4%	26%	4%	-	-	22%	-	4%	17%	22%	-	-	-	-	4%	-
iText	22	23%	14%	32%	5%	-	-	5%	-	14%	14%	18%	-	-	9%	-	-	-
JabRef	33	24%	21%	39%	3%	-	-	3%	6%	-	24%	24%	-	-	3%	-	-	3%
JEdit	26	15%	-	38%	-	4%	-	4%	4%	12%	15%	23%	-	-	-	-	4%	4%
JFreeChart	35	20%	-	9%	6%	-	-	-	-	-	69%	69%	-	-	3%	-	-	-
JFtp	6	-	17%	-	17%	-	-	-	-	17%	17%	33%	-	-	-	-	17%	-
JHotDraw	65	29%	5%	17%	3%	-	-	5%	6%	11%	6%	15%	-	-	25%	-	-	2%
JUnit	23	9%	9%	26%	-	-	4%	22%	4%	4%	17%	22%	-	-	4%	-	-	-
LWJGL	23	13%	9%	39%	4%	-	-	-	-	-	26%	30%	-	-	13%	-	-	-
PMD	42	33%	10%	24%	2%	2%	-	5%	-	-	17%	14%	-	-	10%	2%	-	-
Stripes	18	28%	6%	39%	6%	-	-	-	-	11%	6%	11%	-	-	-	6%	-	-
SweetHome3D	7	57%	-	14%	-	-	-	-	-	-	14%	29%	-	-	-	-	-	-
TvBrowser	58	22%	14%	24%	5%	2%	-	3%	-	2%	14%	24%	-	-	2%	2%	2%	2%
Wicket	83	53%	1%	8%	1%	1%	1%	7%	-	4%	18%	24%	-	-	2%	-	-	-
avg		**24%**	**9%**	**24%**	**4%**	**1%**	**0%**	**5%**	**2%**	**5%**	**20%**	**25%**	**-**	**-**	**6%**	**1%**	**2%**	**1%**

A. *Package–Cluster Similarities*

Our approach is based on the assumption that comparing the modularization of a software system to only one clustering result provides only a limited view. We argue that, as supported by our visualization, only a comparison to multiple clustering results provides a clear picture of criteria used for constructing the package structure. We first check this assumption and provide evidence that several clustering results give meaningful insights. To this end, we identify the best matching cluster of every package according to Equation 1a. We perform this analysis on all 16 projects. Table II summarizes the results by listing the percentage of best matches for each of the 17 concepts of coupling (i.e., a value of 20% means that a fifth of the packages of the respective project is best matched by the respective clustering result).

The results indicate that not only one type of clustering receives all best matches, but the matches are spread across several clustering results. On average, clusters based on structural dependencies have often a high agreement with the package structure (SD.Inh: 24%, SD.Use: 24%) as well as those based on evolutionary coupling (EC.Sup: 20%, EC.Conf: 25%). Together, they already cover most of the matches. The remaining part is spread across all other clustering results, except for CO.Bin, CO.Prop, and CC.II, which do not contain any best matches. Hence, indeed this confirms our assumption that, for identifying modularization patterns, one should consider multiple hierarchies. These results also align with previous results that the congruence of coupling information and the package structure of software systems is not dominated by a single type of coupling [12].

As a further result, showing all 17 clusterings in the visualization is not necessary because some have only little contribution on explaining the package structure. In particular, we rule out the three metrics that do not show any best matches (CO.Bin, CO.Prop, CC.II). Furthermore, we exclude the external fan-out couplings (FO.InhE, FO.AggE, FO.UseE), but include the internal ones (FO.InhI, FO.AggI, FO.UseI), which slightly perform better. Similarly, we exclude SS.LSI and keep SS.Tfidf for semantic similarities. For the following analysis and in all figures of this paper, we hence only use the following set: SD.Inh, SD.Agg, SD.Use, FO.InhI, FO.AggI, FO.UseI, EC.Sup, EC.Conf, CC.I, and SS.Tfidf.

B. *Pilot Examples*

As pilot application examples, we choose *PMD*, a source code analysis tool, and *JHotDraw*, a charting library. They have considerable size (42 and 65 packages) and represent two different kinds of applications. In the following, we use shortened package names, leaving out the package stems `net.sourceforge.pmd` and `org.jhotdraw`. Figures 1 and 4 show the discussed datasets for *PMD* and *JHotDraw*; these images provide previews, but we refer to the interactive version of the tool to follow the interactive exploration process. In particular, brushing and linking as well as selections are required for making some of the discussed observations.

PMD—An example of a *perfect match* (P1.1) is Package `dcd` (*dead code detector*), which is equivalent to clusters from evolution coupling (Cluster A.K of EC.Sup and Cluster A.C of EC.Conf). Either the package was created quite independently from the other code or it has been moved from elsewhere to the current location—the matrix representation of EC.Conf (Figure 3) suggests the second scenario because the cluster produces a dense block of connections on the diagonal and a gradual development usually leads to sparser structures. Moreover, Package `dcd` is also well-matched to Cluster E.A of SD.Use (83%), indicating that it is as well quite cohesive regarding its usage dependencies, which can be confirmed in the respective matrix. Other *perfect matches* (P1.1) are Packages `parsers`, `renderers`, and `util.viewer`. While the latter is again matched by evolutionary coupling (EC.Sup/EC.Conf), the first two have an equivalent cluster in direct inheritance dependencies (SD.Inh)—the criterion to design these packages has been dependencies in the inheritance structure of the contained classes. In addition to those, many other packages are also well-matched: For example, Package `cpd` (*copy/paste detector*) is very similar to Cluster A.C of EC.Sup and Cluster A.D of EC.Conf (88%) as well as to Cluster D of SD.Use (85%). While the first two are rather *addition matches* (P1.2), the last one is an example of a *removal match* (P1.3); further, the match to SD.Use can be classified as a *subdivision match* (P1.4) because the clustering suggest a finer substructure that is not yet reflected by the package structure. Developers could use these insights to slightly adapt or refine Package `cpd`. There are only few examples of clear *partial matches* in *PMD*—most packages are either *matches* or *misses*, which is indicated by the high-contrast, light–dark package structure. One of the few partial matches is Package `util`, which we consider as a *confirmatory split* (P2.4). A 37% agreement with evolutionary coupling (EC.Sup/EC.Conf) shows a split into one bigger cluster and a few smaller ones. In particular, the bigger cluster is already realized in Package `util.viewer`, which has been identified as a *perfect match* already (see above). Package `jsp.ast`, is partially matched by Cluster C from SD.Use, can be considered as a *confirmatory merge* (P2.2) because it suggest to add elements already contained in parent Package `jsp`. Examples of *misses* (P3) are Packages `rules.design`, `rules.strings`, and `symboltable`.

JHotDraw—In contrast to *PMD*, medium and light gray dominate the package structure, indicating less *matches*, but more *partial matches* and *misses*. The only bigger package classified as a *match* is Package `draw.tool`, by trend, an *addition match* (P1.2) to direct inheritance dependencies (SD.Inh). The biggest package among the *partial matches* is Package `draw`, having a 59% agreement to Cluster A of indirect usage dependencies (FO.UseI). The matrix of FO.UseI confirms a clearly cohesive coupling of the classes contained in Cluster A. We classify this pattern as a *suggested merge* (P2.1) because the cluster proposes a union with Package `geom` and parts of Package `samples`. While it is reasonable to merge (or add a common parent package) for Packages `draw` and `geom`, it does not make sense to move elements of Package `samples` because it contains sample code that illustrates the usage of the framework but was intentionally separated from the main framework code. None of the clusterings is able to match this—Package `samples` is classified as a *miss* (P3). Package `app` with an agreement of 53% to SD.Use and FO.UseI is a *suggested split* (P2.3) because both clusterings indicate a subdivision of the package that is not reflected yet. At the same time the clusterings suggest a merge, but again to parts of Package `samples`. In general, leaving out samples from the clustering would improve the matches between the package structure and the clusterings. As a second bigger package, Package `gui` needs to also be classified as a *miss*

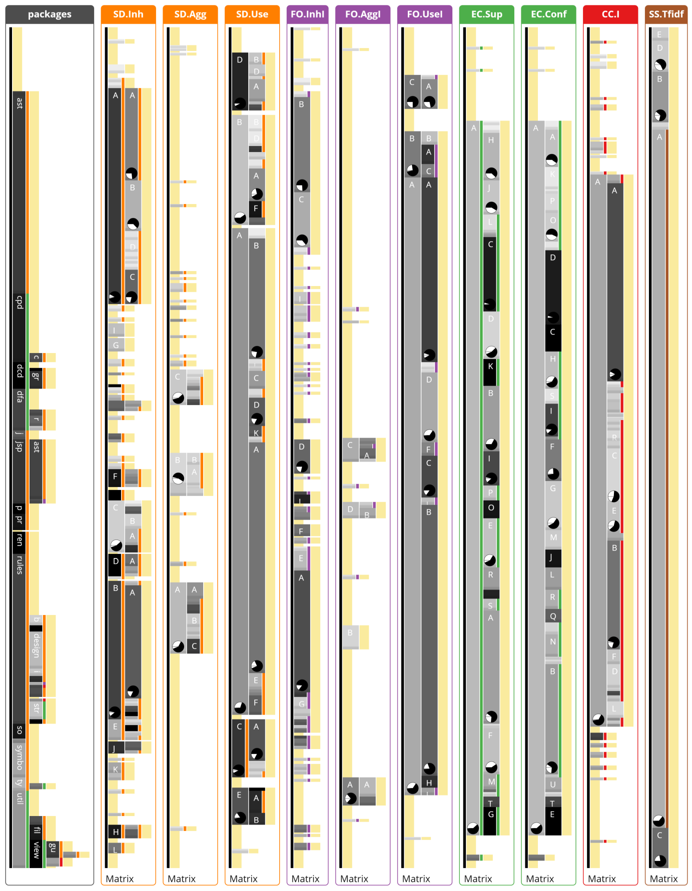


Fig. 4. Visualization of project *JHotDraw*.

(P3). Like visible in the matrix, the coupling information for its elements is too sparse to produce meaningful clusters.

C. Identified Modularization Patterns

After analyzing the pilot examples and defining modularization patterns based on these examples, we expanded our analysis to all 16 software projects contained in the studied dataset [12]. For every project, we investigated its *main packages* that we define as packages containing at least 10 classes and being split at the first non-trivial level of the package structure (or, if the first level is dominated by a single package, the second level of the dominating package). In total, the 16 projects contain 107 main packages. For each of these packages, we decided if it is either a *match*, *partial match*, or *split*. We further discerned subpatterns according to Section IV-B and connect each to one of the clustering results. In some cases, we needed to assign multiple subpatterns because either (a) subpatterns might occur together for the same package–cluster match or (b) multiple clusters matched to a very similar extent. Table III reports our findings.

Considering the main patterns, 25% of the packages (27 of 107) are classified as *matches*. For these, it is quite clear to which clustering they relate to, and hence, what was a criterion to construct them. For *matches*, we discerned 1.8 subpatterns per package, mostly *perfect matches* (P1.1), *addition matches*

TABLE III
NUMBER OF IDENTIFIED PATTERNS OF ALL MAIN PACKAGES AMONG ALL 16 PROJECTS DISCERNED BY ORIGIN OF THE MATCHED CLUSTERS.

	SD			FO			EC		CC	SS
	Inh	Agg	Use	InhI	AggI	UseI	Sup	Conf	I	Tfidf
P1					**27**					
P1.1	3	-	1	1	-	-	3	3	-	-
P1.2	4	-	2	-	-	1	5	5	-	-
P1.3	3	1	3	-	-	1	6	6	-	-
P1.4	-	-	1	-	-	-	-	-	-	-
P2					**58**					
P2.1	6	2	8	1	1	8	9	11	1	-
P2.2	-	-	-	-	-	-	-	-	-	-
P2.3	12	4	12	4	1	9	12	14	1	-
P2.4	3	-	4	-	-	-	-	-	-	-
P3					**22**					

(P1.2), and *removal matches* (P1.3), but rarely *subdivision matches* (P1.4). The *matches* are frequently related to some of the structural dependencies and evolutionary coupling (SD.Inh, SD.Use, EC.Sup, and EC.Conf). Hence, these metrics seem to play an important role for designing packages.

Partial matches form the biggest group of patterns, with 54% of the packages (58 of 107). In particular, *suggested merges* (P2.1) and *suggested splits* (P2.3) occur frequently. We observed the two patterns often together—a cluster suggests to merge other classes into the package and would remove others. This co-occurrence is also reflected in the higher number of 2.1 subpatterns per package. In isolation, *suggested splits* are more frequent than *suggested merges*. This bias might be influenced by our focus on larger packages—a merge would require even larger clusters. We also identified a few *confirmatory splits* (P2.4), but no *confirmatory merges*. Similarly, this imbalance is likely an artifact of our data acquisition: *confirmatory merges* are only possible on the second level of the package structure and below. Regarding the clusterings, again SD.Inh, SD.Use, EC.Sup, and EC.Conf represent the majority of matches, but in addition, SD.Agg, FO.InhI, and FO.UseI have an influence. *Confirmatory matches* only appear for SD.Inh and SD.Use. According to our experience, *structural dependencies* (SD) usually produce fine-grained clusterings in several layers, which makes confirmatory patterns more likely.

Misses only appear in 21% of the packages (22 of 107). Our data and visualization does not provide enough information to explain the criteria for these packages.

We also recorded the durations that the author who performed the analysis required to classify the packages. We exclude *PMD* and *JHotDraw* from this analysis because, as our pilot examples, we studied them in more detail, which took longer. For each of the remaining 14 projects, it took 1–9 min to classify the 2–12 main packages. A total of 59 min results in about 4 min per project on average and about 40 s for one of the 91 main packages.

VI. DISCUSSION AND LIMITATIONS

Motivating our approach in Section I, we presented two use cases: Use Case I describes software developers extending or restructuring a software systems and Use Case II outlines a research scenario where software engineering scientists study modularization patterns. The presented results (Section V) now provide examples of both scenarios. First, analyzing an individual project and studying modularization patterns of central packages is demonstrated in the pilot examples (Section V-B) and matches Use Case I. Second, broadening the analysis to a larger set of projects studying common patterns as done in Section V-C is already a specific instance of Use Case II. Hence, the approach is not only a tool that can be leveraged in practical software development, but equally in research to explore modularization criteria and generate hypotheses for further investigation.

Whereas these applications reflect realistic examples, their ability to provide evidence for the usefulness of our approach is limited: First, the interactive visualization approach was only used by the authors themselves. Hence, the results only reflect what trained users might be able to find, but it remains unknown how self-explaining and easy to use the tool is for untrained users. Furthermore, we did not contrast the approach to another technique because it is unclear what is an appropriate baseline for this scenario. It is even difficult to find a visualization tool that could represent the same data—some other hierarchy comparison approaches do (cf. Section II), but they are not tailored for a software engineering scenario. Software engineering tools, in contrast, lack the ability to provide an overview on and compare multiple clustering results. Finally, the approach has only been tested for small to medium size open source Java systems.

VII. CONCLUSIONS AND FUTURE WORK

We presented a novel approach for comparing a modularization of a software system to multiple clustering results. By juxtaposing icicle plots and encoding similarities in colors as well as making them interactively explorable, users are enabled to identify modularization patterns. These patterns might explain the criteria that were used to construct a module or hint at weaknesses in the modularization. By providing realistic examples, we show that the patterns could lead to important insights for extending and remodularizing a software system as well as for studying modularization in general.

As part of future work, we consider extending our visual comparison approach towards interactive editing of a hierarchical software modularization. This has been investigated in context of remodularizing software systems for individual hierarchies [41], pairs of hierarchies [22], or extracting specific components [18], but not for the comparison of the current modularization to a set of clustering results. Multiple automatically generated modularizations could much better support a software developer to find appropriate improvements and to understand the effects of a remodularization. Also, we want to test our approach with untrained users to find strengths and weaknesses in the usability and applicability of our approach.

Acknowledgment

Fabian Beck is indebted to the Baden-Württemberg Stiftung for the financial support of this research project within the Postdoctoral Fellowship for Leading Early Career Researchers.

References

[1] J. Kruskal and J. Landwehr, "Icicle Plots: Better displays for hierarchical clustering," *The American Statistican*, vol. 37, no. 2, 1983.

[2] 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*. IEEE Computer Society, 1999, pp. 50–59.

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

[4] A. Wierda, E. Dortmans, and L. L. Somers, "Using version information in architectural clustering – a case study," in *Proceedings of the Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 2006, pp. 214–228.

[5] O. Maqbool and H. Babri, "Hierarchical clustering for software architecture recovery," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 759–780, 2007.

[6] B. S. Mitchell and S. Mancoridis, "On the evaluation of the Bunch search-based software modularization algorithm," *Soft Computing*, vol. 12, no. 1, pp. 77–93, 2007.

[7] A. Kuhn, S. Ducasse, and T. Girba, "Enriching reverse engineering with semantic clustering," in *Proceedings of the 12th Working Conference on Reverse Engineering*. IEEE Computer Society, 2005, pp. 133–142.

[8] L. Voinea and A. Telea, "CVSgrab: Mining the history of large software projects," in *Proceedings of the Joint Eurographics - IEEE VGTC Symposium on Visualization*. Eurographics Association, 2006, pp. 187–194.

[9] A. Vanya, L. Hofland, S. Klusener, P. Van De Laar, and H. Van Vliet, "Assessing software archives with evolutionary clusters," in *Proceedings of the 16th IEEE International Conference on Program Comprehension*. IEEE Computer Society, 2008, pp. 192–201.

[10] L. L. Silva, M. T. Valente, M. de A Maia, and N. Anquetil, "Developers' perception of co-change patterns: An empirical study," in *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2015, pp. 21–30.

[11] F. Beck and S. Diehl, "On the impact of software evolution on software clustering," *Empirical Software Engineering*, vol. 18, no. 5, pp. 970–1004, 2013.

[12] ——, "On the congruence of modularity and code coupling," in *Proceedings of the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering and 13th European Software Engineering Conference*. ACM, 2011, pp. 354–364.

[13] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "An empirical study on the developers' perception of software coupling," in *Proceedings of the 35th ACM/IEEE International Conference on Software Engineering*. IEEE, 2013, pp. 692–701.

[14] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl, "A reverse engineering approach to subsystem structure identification," *Journal of Software Maintenance: Research and Practice*, vol. 5, no. 4, pp. 181–204, 1993.

[15] M. Glorie, A. Zaidman, A. van Deursen, and L. Hofland, "Splitting a large software repository for easing future software evolution—an industrial experience report," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 21, no. 2, pp. 113–141, 2009.

[16] G. M. Rama, "A desiderata for refactoring-based software modularity improvement," in *Proceedings of the 3rd India Software Engineering Conference*. ACM, 2010, pp. 93–102.

[17] H. Washizaki and Y. Fukazawa, "A technique for automatic component extraction from object-oriented programs by refactoring," *Science of Computer Programming*, vol. 56, no. 1-2, pp. 99–116, 2005.

[18] A. Marx, F. Beck, and S. Diehl, "Computer-aided extraction of software components," in *Proceedings of the 17th Working Conference on Reverse Engineering*. IEEE, 2010, pp. 183–192.

[19] C. McMillan, N. Hariri, D. Poshyvanyk, J. C. Huang, and B. Mobasher, "Recommending source code for use in rapid software prototypes," in *Proceedings of the 2012 International Conference on Software Engineering*. IEEE, 2012, pp. 848–858.

[20] M. Hall, N. Walkinshaw, and P. McMinn, "Supervised software modularisation," in *Proceedings of the International Conference on Software Maintenance*. IEEE, 2012, pp. 472–481.

[21] D. Holten and J. J. van Wijk, "Visual comparison of hierarchically organized data," *Computer Graphics Forum*, vol. 27, no. 3, pp. 759–766, 2008.

[22] R. Lutz, D. Rausch, F. Beck, and S. Diehl, "Get your directories right: From hierarchy visualization to hierarchy manipulation," in *Proceedings of the 2014 IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE, 2014, pp. 25–32.

[23] F. Beck and S. Diehl, "Visual comparison of software architectures," *Information Visualization*, vol. 12, no. 2, pp. 178–199, 2013.

[24] F. Beck, F.-J. Wiszniewsky, M. Burch, S. Diehl, and D. Weiskopf, "Asymmetric visual hierarchy comparison with nested icicle plots," in *Joint Proceedings of the Fourth International Workshop on Euler Diagrams and the First International Workshop on Graph Visualization in Practice*, 2014, pp. 53–62.

[25] A. Telea and D. Auber, "Code Flows: Visualizing structural evolution of source code," *Computer Graphics Forum*, vol. 27, no. 3, pp. 831–838, 2008.

[26] C. Vehlow, F. Beck, and D. Weiskopf, "Visualizing dynamic hierarchies in graph sequences," *IEEE Transactions on Visualization and Computer Graphics*, 2016.

[27] M. Graham and J. Kennedy, "A survey of multiple tree visualisation," *Information Visualization*, vol. 9, no. 4, pp. 235–252, 2010.

[28] S. Bremm, T. von Landesberger, M. Hess, T. Schreck, P. Weil, and K. Hamacherk, "Interactive visual comparison of multiple trees," in *Proceedings of the 2011 IEEE Conference on Visual Analytics Science and Technology*. IEEE, 2011, pp. 31–40.

[29] M. Graham and J. Kennedy, "Extending taxonomic visualisation to incorporate synonymy and structural markers," *Information Visualization*, vol. 4, no. 3, pp. 206–223, 2005.

[30] F. Chevenet, C. Brun, A. L. Banuls, B. Jacq, and R. Christen, "TreeDyn: towards dynamic graphics and annotations for analyses of trees," *BMC Bioinformatics*, vol. 7, no. 1, pp. 439+, 2006.

[31] E. R. Tufte, *The visual display of quantitative information*. Graphics Press, 1983.

[32] K. Sugiyama, S. Tagawa, and M. Toda, "Methods for visual understanding of hierarchical system structures," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 11, no. 2, pp. 109–125, Feb 1981.

[33] T. Munzner, F. Guimbretière, S. Tasiran, L. Zhang, and Y. Zhou, "TreeJuxtaposer: Scalable tree comparison using focus+context with guaranteed visibility," *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 453–462, 2003.

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

[35] J. Laval, S. Denier, S. Ducasse, and A. Bergel, "Identifying cycle causes with enriched dependency structural matrix," in *Proceedings of the 16th Working Conference on Reverse Engineering*. IEEE, 2009, pp. 113–122.

[36] M. Greilich, M. Burch, and S. Diehl, "Visualizing the evolution of compound digraphs with TimeArcTrees," *Computer Graphics Forum*, vol. 28, no. 3, pp. 975–982, 2009.

[37] M. Burch, C. Vehlow, F. Beck, S. Diehl, and D. Weiskopf, "Parallel Edge Splatting for scalable dynamic graph visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 12, pp. 2344–2353, 2011.

[38] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Systems Journal*, vol. 13, no. 2, pp. 115–139, 1974.

[39] 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.

[40] M. Rosvall and C. T. Bergstrom, "Multilevel compression of random walks on networks reveals hierarchical organization in large integrated systems," *PLoS ONE*, vol. 6, no. 4, p. e18209, 2011.

[41] M. Beck, J. Trümper, and J. Döllner, "A visual analysis and design tool for planning software reengineerings," in *Proceedings of the 6th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE, 2011, pp. 1–8.