Visual Comparison of Software Architectures

Fabian Beck
beckf@uni-trier.de
University of Trier, Germany

Stephan Diehl
diehl@uni-trier.de
University of Trier, Germany

September 18, 2012

Abstract

Reverse engineering methods produce different descriptions of software architectures. In this article we address the task of exploring and comparing these descriptions. We present a novel visualization technique to compare architectures consisting of a decomposition of the software system and the dependencies among the code entities. This technique uses a visual representation of an adjacency matrix to provide a scalable analysis tool. Advanced layout features like an automatic level of detail algorithm and sorting strategies improve the readability of the visualization. In a case study, we show how this technique can be applied in practice.

Keywords: Software architecture; hierarchy comparison; graph visualization

Appeared in Information Visualization; first published September 17, 2012; doi: 10.1177/1473871612455983. The final version can be found on the publisher's site.

1 Introduction

Understanding the architecture of a software system is important for maintaining and evolving the system. The architecture is often described in manually created documents and diagrams. But there is no guarantee that these files match the architecture that is actually implemented. The only reliable data source of this factual architecture is the source code itself. But it contains the architecture just implicitly—in form of the code structure and its dependencies.

There exist different methods to extract the implicit architecture from the code. We can just take the directory or package structure of a project. We might ask an expert to manually decompose the system.[1] Or we apply a software clustering algorithm[2] to generate a hierarchical structure of the source code. These methods provide different decompositions of the system as a partial description of its architecture.

Dependencies between code entities reflect another important part of the architecture. For instance, inheritance represents a dependency between two classes, while method calls form dependencies between methods. Dependencies can be retrieved from the static source code or observed dynamically at runtime. But there could also be hidden dependencies: Two code entities might be related by their common evolution—they might have changed together frequently.[3] Documenting these dependencies on a high level of abstraction also makes implicitly contained architecture information explicit.

There are many possible descriptions of an architecture in form of different software decompositions and dependency types—there does not exist *the* architecture description of a system. It is important to compare these different descriptions because this

- could approach the factual architecture of the system,
- might hint at high-level differences between two versions, for instance, detecting architectural drifts, and
- may help creating more reliable architecture descriptions.

For instance, comparing the initial architecture of a system to the architecture at a later point of development could reveal architectural drifts. Checking the implemented architecture against the documented one may identify architecture violations. Or contrasting the architectures automatically extracted by different algorithms might enable us to combine the advantages of the algorithms.

There are many tools that visualize software architectures.[4] But these tools only show one description of an architecture. The goal of this work is to develop a visualization approach that enables comparing two software architectures, each consisting of a hierarchical decomposition of the system and a dependency graph. Our approach is based on an adjacency matrix representation of the graphs with attached hierarchy diagrams. Visual features and interactions support the users in finding similarities and differences in the hierarchical structures as well as in the graphs.

At first, we need to identify concrete tasks that support such a comparison of software architectures (Section 2). We introduce our visualization technique to explore

and compare architectures based on software decompositions and code dependencies (Section 3) and present advanced layout features (Section 4). In a case study we apply this visualization technique to analyze package structures and clustering results (Section 5). Finally, we compare our approach with related techniques (Section 6), discuss application scenarios (Section 7), and draw some conclusions (Section 8).

This paper is an extended version of a paper presented at SoftVis '10.[5] The main extensions are improved interactions (Section 3.3), an evaluation of the adaptive level of detail algorithm (Section 4.1.4), an elaborate global sorting strategy (Section 4.2.2), and a broader discussion of possible application scenarios for the presented visualization approach (Section 7).

2 Comparing Architectures

In a previous study[6] we compared the capabilities of different data sources to recover the architectures of software systems. In particular, we used different dependency types and applied a clustering algorithm that produced a hierarchical decomposition of the system. To assess the quality of the automatically generated decompositions, we had to compare them to a reference decomposition. There exist different metrics that implement such a comparison of software decompositions[2]—we used *MoJoFM*,[7] a metric that counts the minimal number of move and join operations necessary to transform one decomposition into the other. This metric-based approach solved the problem of assessing the quality of the decompositions but left some questions unanswered:

- What are the matching and non-matching parts of the decompositions?
- What are the reasons for clustering together certain parts of the software project?
- How can we explain the different results when applying the algorithm to different software projects?

Starting from these questions, we felt that there is a need for understanding the differences of software decompositions and code dependencies. The experience gained from our study now helps us to identify important tasks supporting the comparison of software architectures. Later, we return and apply our visualization technique to the data set of this previous study (Section 5).

2.1 Decompositions & Dependencies

Our work is based on extracting the architecture of the software system from source code. The extracted architecture consists of two parts: the decomposition of the system and the dependencies among the parts of the system.

To provide a widely applicable visualization, we do not limit those data structures to particular concepts, such as components and connectors or classes and inheritance. Only as an example, we use classes that are organized in packages connected by different types of dependencies in this paper. As long as the data matches the following definitions, it can be visualized using our visualization technique.

We call the elementary code units of the system *code entities*. Depending on the particular application, these entities could be methods, classes, packages, or components. Let V be the set of all code entities. The *dependency structure* of the system is a directed graph on the set of code entities $G = (V, E_G)$ where the set of edges $E_G \subseteq V \times V$ represents the dependencies between the entities V. A *software decomposition* divides these entities into groups or clusters of entities, which are usually hierarchically organized (e.g., in a package structure or as a result of a hierarchical clustering algorithm). Thus, a *software decomposition* is a hierarchy (i.e., a tree) $H = (\hat{V}, E_H)$ where $\hat{V} = V \cup C$ consists of all code entities V and all clusters C, and the tree edges $E_H \subset \hat{V} \times \hat{V}$ express the containment relation such that V contains all leaf nodes and C all intermediate nodes of the hierarchy H. In terms of graph theory, such a combination of a graph and a hierarchy is called *compound graph*.

Since our approach aims at the comparison of these data structures, we want to contrast at least two such compound graphs on the same set of code entities.

Clustering algorithms need characteristic data about the artifacts to cluster. Usually, this data is expressed either as the similarity of artifacts in a similarity matrix or as dependencies between the artifacts in a dependency graph. Since our previous study uses a graph-based clustering approach, we will focus on the latter case of dependency data. But a similarity matrix is equivalent to a weighted graph where all possible edges exist, a complete graph.

2.2 Tasks

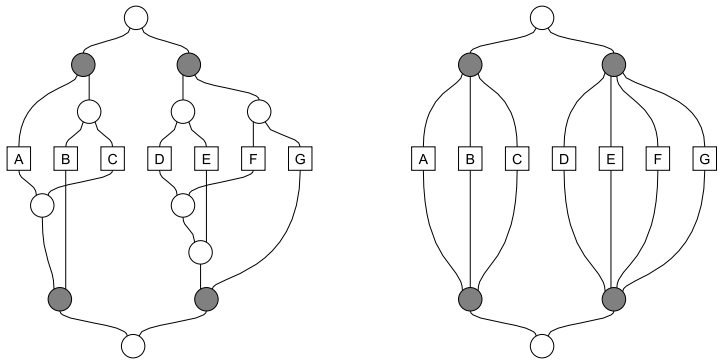
Before actually designing a tool that supports a user to compare software architectures based on software decompositions and code dependencies, we first need to analyze the comparison process in greater detail. To this end, we will identify key tasks. Since the user should be able to solve these tasks, they form the requirements for a comparison tool. Following the taxonomy of Chikofsky and Cross,[8] the tasks are part of *reverse engineering* (in particular, *design recovery*) and *restructuring* on the abstraction level of design.

Task 1 *Analyze and compare different types of code dependencies.*

When we only look at the dependency structure of a system, some interesting questions already arise. For instance, dependency information might only sparsely cover the entities, or there might be clusters of entities, outliers, or hubs. Such characteristics emerge, in particular, when comparing different dependency types. For instance, static structural dependencies like method calls can be compared with dynamic dependencies. One type of dependencies provides points of reference to better analyze the other.

Task 2 *Relate a software decomposition to the dependency structure.*

Software decompositions are supposed to follow the concept of high cohesion and low coupling:[9] The code entities of a cluster should be linked by many dependencies (high cohesion) whereas there should only be few dependencies that cross cluster borders (low coupling). Thus, a decomposition of the software might be closely related to



(a) (b)

Figure 1: Two different decompositions on the same set of code entities: (a) totally expanded decompositions; (b) gray clusters collapsed.

the dependency structure. Making the connection between both explicit, we may find clusters of dependencies that explain why certain code entities are grouped together. For instance, a part of the system belongs together due to many method calls that connect the code entities it contains. Moreover, we may compare the cohesion of a cluster to its coupling to other clusters. Such information might explain the relation between a dependency type and the decomposition. If the decomposition is created automatically, it could show how the dependencies influence the clustering results.

Task 3 *Compare different software decompositions at a matching level of detail.*

Finally, we look at the different software decompositions. Two decompositions are similar if the clusters of one decomposition match the clusters of the other decomposition. Metrics like MoJoFM[7] can be used to exactly measure this similarity. But finding out which clusters actually match each other and which clusters do not have any match in the other decomposition is more interesting from the perspective of a software engineer who, for instance, wants to restructure the system.

Another aspect is the level of detail of a decomposition. The hierarchical structure of the decompositions allows considering clusters on different levels. The two decompositions presented in Figure 1 (a)—one above the entities, one below—look significantly different at first glance. But when collapsing particular clusters of the decompositions as depicted in Figure 1 (b), the partition of code entities is identical in both decompositions. Finding such matching levels of detail might, however, be difficult for larger decompositions. A tool that supports the user in this process would be necessary.

With respect to comparing software decompositions, we require such a tool to show similarities and differences of decompositions and to support finding matching levels of detail.

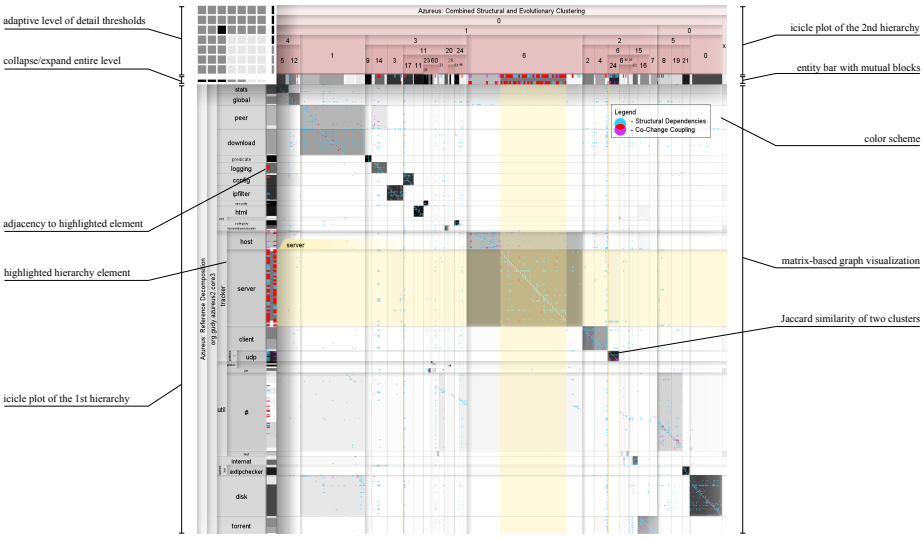


Figure 2: Example of the novel matrix-based visualization technique to compare different dependency graphs and software decompositions.

Since these tasks focus on an explorative, qualitative—not quantitative—analysis, we believe visualization is most suitable. A metric-based approach or a textual representation would not provide sufficient overview and flexibility.

3 Visualization Technique

A visual technique that supports the user to solve the three tasks is required

- to concurrently display dependency graphs and software decompositions,
- to reveal similarities and differences in graphs and decompositions, and
- to support finding matching levels of detail in different software decompositions.

To simplify these requirements somewhat, we decided to only allow two dependency graphs and two software decomposition at maximum. Nonetheless, multiple comparisons could be realized by several pairwise comparisons.

Figure 2 provides a preview of how our novel visualization looks. It shows a representation of the Azureus (now called Vuze) system, a BitTorrent client. The following sections introduce the visualization step by step. We discuss the representation of the dependency graphs and the software decompositions, introduce a metric to find matching clusters, and explain the interactive features of our implementation. Finally, we provide a comprehensive example how to read the resulting images as presented in Figure 2.

3.1 Dependency Representation

Our visualization technique is based on an adjacency matrix representation of graphs: It represents code entities as rows and columns of a matrix, and it depicts dependencies as cells of the matrix. A colored box at the intersection of row A and column B thus encodes a dependency from code entity A to code entity B. Hence, all code entities are represented twice, once as a row and once as a column. In a usual adjacency matrix, rows and columns are ordered equally so that self-dependencies form the diagonal of the matrix. However, our visualization deviates from this paradigm; we explain later on why.

We preferred a matrix representation over a node-link approach—diagrams where nodes represent the code entities and visual links between these nodes represent code dependencies—for several reasons:

Scalability Node-link diagrams suffer from occlusion problems when it comes to visualizing larger and denser graphs. Elaborate layout algorithms may ease the problem, but cannot eliminate it. In contrast, no visual elements overlap in matrix visualizations by definition. Ghoniem et al.[10] provide empirical evidence for the superiority of matrix representations of larger graphs in many applications.

Edges Since we want to analyze differences in dependency graphs, we are interested in the existence of particular edges (i.e., dependencies). In contrast, tracking paths over several edges and code entities—an obvious shortcoming of matrix-based graph visualizations—is less important for our application because path related tasks are not required to identify clusters. A matrix visualization focuses on edges; it explicitly shows existing and non-existing edges.

Clusters Depending on a good layout, both node-link and matrix diagrams are able to reveal clusters. But as Henry et al.[11] point out, in dense clusters, matrix representations still provide detailed information while node-link representations produce clutter.

Figure 2 shows the complete visualization consisting of an adjacency matrix as the central part. The matrix is attached with two hierarchy representations and other supplementary diagrams. In this example, we used 477 classes as code entities, which results in a matrix of 477 rows and 477 columns. These rows and columns are too narrow to be indicated by a border line. Thus, each cell representing a dependency (an intersection of a row and column of the matrix) has only a few pixels on screen, but we can still see and discern these small points. Moreover, we observe that these colored cells are not evenly distributed over the image. The visual clusters formed by these cells hint at clusters in the dependency structure.

Our matrix-based approach is able to visualize two graphs on the same set of code entities in the same diagram. The dependencies just need to be drawn in different colors—one color for each graph and a third color to represent duplicate dependencies (i.e., dependencies that occur in both graphs). Figure 2 provides an example with two such types of dependencies. The legend depicts the color scheme: blue for the first type, purple for the second type, and red for the duplicate dependencies. Concurrently

visualizing more than two graphs with this approach is possible, but would probably confuse the user by ambiguous colors. A comparison of n graphs would need $2^n - 1$ different colors plus a background color.

If we work on weighted graphs instead, the matrix-based representation needs some adaptations: Since we already used the color to encode the different data sources, we have to encode the values of the similarity metric otherwise. We decided to use opaqueness and draw weaker edges less opaquely than stronger ones, which is very intuitive. For edges included in both graphs, we sum up the weights of both edges and accordingly choose the degree of opaqueness.

3.2 Decomposition Representation

We consider software decompositions as hierarchies. A visual representation of a hierarchy can be easily attached to the sides of the matrix. We use a layered icicle plot[12] to depict this hierarchy. Such an icicle plot lays out the nodes similar to a usual tree diagram, but depicts each node as a box that fills the available space around the node. It is more space-efficient and easier to label than an equivalent tree diagram.

The visualization in Figure 2 displays a software decomposition in form of the package structure on the left hand side of the diagram. Soft shadows separate the clusters, not only in the hierarchy but also continuously in the matrix. If enough screen space is available, labels identify the clusters. We align the leaves of the hierarchy, which represent the code entities, in a bar. This entity bar can be used for displaying additional information on the entities (Section 3.3 and Section 4.2.1)

Since the rows and columns of the matrix can be sorted independently, we are able to add a second software decomposition on top of the diagram. Additional to the package structure on the left, the example in Figure 2 depicts a decomposition automatically generated by a clustering algorithm (details on clustering are explained in Section 3.4).

The hierarchical structure of each decomposition implies some constraints on the order of the code entities: Only sibling entities or clusters are allowed to be switched without destroying the representation of the decomposition. Hence, in the general case, code entities have to be sorted differently with respect to rows than with respect to columns.

3.2.1 Cluster Similarity

The task of comparing the two decompositions consists of finding similarities and differences in the cluster structure. But without assistance, this would be a time-consuming and strenuous task: Considering a particular cluster, it is hard to identify its most similar correspondent because it has to be manually compared to every cluster in the other decomposition. A metric that is able to rank the possible correspondents with respect of their similarity to a selected node might solve the problem.

A cluster consists of a set of code entities. Thus, comparing two clusters is equivalent to comparing two sets A and B. To get a similarity measure, we are interested in how many entities concurrently belong to both clusters in relation to the size of both

clusters. This can be expressed as the size of the intersection of the two sets divided by the size of the union of the sets—the Jaccard coefficient:

$$\mathrm{sim}(A, B) := \frac{|A \cap B|}{|A \cup B|}.$$

We integrate the similarity information based on the Jaccard coefficients in the background of the matrix representation. The clusters form a matrix-like meta-structure where the cluster—not the code entities—represents the rows and columns. Each comparison of two clusters can be represented as a cell of this matrix. We use the background brightness of the cells to encode the Jaccard similarity value of the according cluster: Dark backgrounds visualize high similarity values. Coloring each possible pair of clusters like this would, however, lead to overlapping cells and thus ambiguous shadings. Hence, this approach enables comparing two decompositions only at one level of detail for each decomposition. In our case the background structure always shows the cluster similarity at the lowest levels. But by temporarily collapsing clusters in the hierarchy, this lowest level can be adapted with our tool. By default this is done manually by clicking on the clusters, but we also implemented an algorithm to support the task of finding matching levels (Section 4.1).

3.3 Interaction

Our implementation of the approach follows the information visualization mantra:[13] *overview first, zoom and filter, then details-on-demand.*

Overview: The matrix shows the whole data set without the need to scroll or manipulate the view. Hence, overview is provided by the default view of the visualization at any time. Nevertheless, the similarity metric, which is visualized in the background, only allows comparing the clusters at the lowest level of the hierarchy. To support the comparison of clusters on higher levels, the visualization allows the user to collapse (and expand) clusters by clicking on their visual representations. A collapsed cluster does not change its size, but its subclusters temporarily disappear. The collapsed cluster now directly contains all leaf nodes of the subclusters. Larger gray scale background boxes display the cluster similarity metric values on this higher level. Furthermore, slim markers on the side of the hierarchy enable the user to collapse or expand whole hierarchy levels (Figure 2, top left). These markers also indicate which levels are currently totally collapsed (light gray), partially collapsed (gray), or fully expanded (dark gray).

Zoom: A matrix is a table where zooming can be implemented like demonstrated by the Table Lens visualization using multiple focuses[14]—rows and columns can be focused independently by zooming; a selected row gets higher, a column gets broader. In our visualization, zooming an element of the hierarchy means that the respective rows or columns get larger (or smaller). Zooming is triggered by using the mouse wheel while hovering the cursor on a cluster. This can be done for clusters in the first as well as in the second hierarchy independently. Our goal is that the total size of the visualization does not change while zooming, which is implemented by reducing the size of the remaining clusters accordingly. Zooming different clusters hence creates

Figure 3: By highlighting a package or entity, small colored stripes indicate adjacent entities (detail).

a set of multiple focuses in the matrix: areas where rows are enlarged, columns are enlarged, and both rows and columns are enlarged.

Filter: Discerning the three different colors that encode the type of an edge gets harder for larger graphs when the respective cell only consists of a few pixels on screen. Interactively switching on and off the edges of a certain color facilitates this comparison in a more scalable way. Additionally, some details presented on demand include a certain filtering aspect as the following paragraph explains.

Details on demand: When moving the mouse over a colored matrix cell representing a dependency, the labels of both related code entities appear. Moving the mouse over a code entity or a cluster, the tool shows its name as a tooltip. Besides these basic features, it is possible to filter the graph structure on demand (Figure 3): When an entity or cluster is hovered, all adjacent entities are highlighted at the leaf level in the entity bars of the hierarchies. Small colored stripes indicate the adjacent entities while their color encodes the type of adjacency. The alignment of the stripes (left or right in the vertical entity bar; top or bottom in the horizontal entity bar) discerns incoming edges from outgoing edges.

3.4 Application Example

Next, we provide a first application example to give an impression on how to use the introduced visualization technique (this example is illustrated in more detail in the supplementary materials). Figure 2 depicts the 477 classes of the core of the *Azureus* project in our matrix visualization approach. It shows the package structure of the system on the left and a hierarchical structure derived by clustering on the top. The clustering was created by the graph-based software clustering tool Bunch.[15] The tool follows a search-based approach and optimizes a clustering quality metric by using a hill climbing algorithm. Applying the clustering algorithm recursively, a hierarchy of clusters is created. The two graphs encode two types of dependencies automatically detected in the project. The first type consists of structural static code dependencies (method calls, aggregation, inheritance, etc.). The latter one is represented by co-change couplings (also known as logical couplings or evolutionary couplings). These are couplings that indicate that two classes have been changed frequently together in the past. The union of both graphs, which is defined as the union of the sets of edges, is used as the input for the clustering algorithm.

The visualization enables the user to identify similar clusters at a glance: With the help of the background structure, we can immediately detect the most similar cluster combinations. Moreover, non-matched clusters result in rows or columns consisting only of a set of light-gray boxes without any darker ones. Table 1 lists some examples of such matched and non-matched clusters. For instance, the dark box in the lower right corner in Figure 2 shows that the *disk* package nearly matches the *0.0.0* cluster. The only light-colored boxes in the row of the *util.#* package provide an example of

a non-matched package. But there are significant differences between types of non-matched clusters: While the *util.#* package is far from having a matching counterpart, for instance, the *0.1.1* cluster is just mainly distributed over two packages (*peer* and *download*). This union of two packages in one of the clusters, which is indicated by two horizontal mid-gray boxes, suggests that the *peer* and *download* packages are related and that an aggregation to a common parent package may improve the architecture of the software. Analogously, two vertical mid-gray boxes are an indicator that splitting a package could be recommendable in this application scenario (e.g., the *client* package).

Table 1: Examples of matched and non-matched clusters in Figure 2.

Package Decomposition	Clustered Decomposition
disk	0.0.0
tracker.prot.udp	0.1.2.6.24
ipfilter	0.1.3.3
util.#	–
–	0.1.1

The graphs consist of 2362 edges (structural) and 302 edges respectively (co-change). The size of the data set prohibits encoding an edge in much more than one pixel on the screen. Nevertheless, we can get a rough overview on the two graphs by just looking at the visualization presented in Figure 2. When we want to retrieve details on the graph structure, we may zoom in on one of the packages or clusters. For instance, in Figure 2, the *tracker.server* package and the *0.1.6* cluster are enlarged. An alternative way to reveal detailed information, is to use the detail-on-demand and filtering functionality as also demonstrated in Figure 2: The *tracker.server* package is also highlighted, which not only marks all contained entities but also shows the adjacency relations of these contained entities in the entity bars of the hierarchies as colored stripes as explained in Section 3.3 and Figure 3. In this example, we are able to state that the *tracker.server* package is significantly related to the *logging*, *tracker.host*, *tracker.prot.udp*, and *util* packages. We can even retrieve the direction of the dependency by the alignment of the stripes: Mostly the *tracker.server* package depends on the mentioned other packages (not vice versa) because the stripes are aligned to the left border of the entity bar (see also Figure 3).

4 Advanced Layout

In the basic visualization technique described so far, there is room for some advanced layout improvements. One aspect is the level of detail of the two hierarchies—in large hierarchies it could be hard to manually find a matching level of both hierarchies. Another aspect concerns the ordering of the vertices—an optimized order might reveal additional insights. Next, we will introduce an adaptive level of detail algorithm and two sorting strategies and demonstrate the improvements in two small evaluations. In Figure 2 we already applied these improvements to provide a more readable visualization.

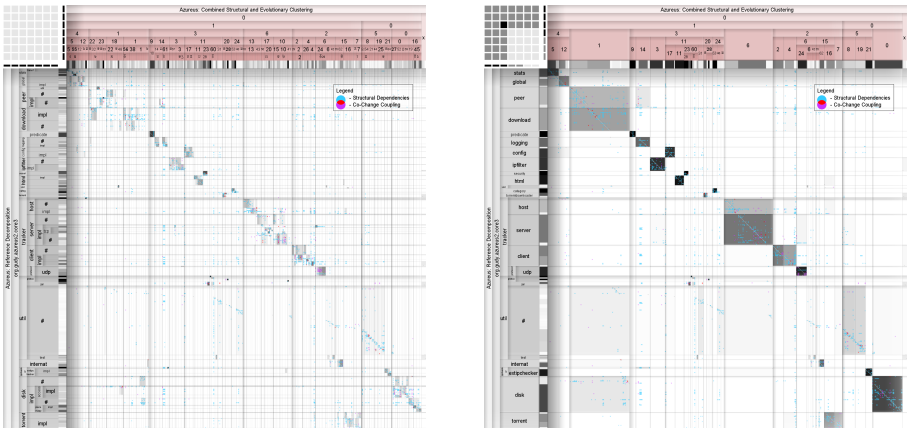


Figure 4: The example from Figure 2 without and with an appropriately chosen level of detail.

4.1 Adaptive Level of Detail

When comparing two decompositions, it is necessary to choose an appropriate level of detail. The gray scale matrix in the background is the most important criterion to assess the similarity of the two decompositions at a particular level. Roughly speaking, few black boxes and many white boxes indicate a high conformance while many low-contrast gray boxes indicate a low conformance. The interactive expand and collapse mechanism allows the user to explore different levels, but for larger data sets this could become tedious: Clustered decompositions tend to be deep and fine-grained hierarchies while, for instance, package structures are normally flat and more coarse-grained. Hence, an automatic or semi-automatic algorithm that helps finding two matching levels-of-detail would be of great help.

We define a level of detail to be appropriate if the following conditions are true:

Conformance The decompositions match as far as possible with respect to a measure of similarity.

Significance The structure of both decompositions is preserved (i.e., not too many clusters should be collapsed).

It is always possible to reach maximum conformance by totally collapsing both decompositions. But this obviously violates the condition of significance. Hence, these two conditions usually must be traded off against each other.

Figure 4 gives an example of how important the level of detail is. While, in the default visualization on the left hand side, the background patterns are much too fine-grained to easily find differences and similarities in both decompositions, the right hand side image is much more readable because it has an appropriately chosen level of detail.

4.1.1 Optimization Criterion

To implement an automatic algorithm, we had to find a formal optimization criterion that assesses the quality of a particular level of detail state. Such a state consists of the partially collapsed decompositions. Each collapsed decomposition implies a partition of the code entities like presented in the example of Figure 1. Hence, an optimization criterion is a real-valued objective function defined on two partitions.

We propose an objective function that counts the number of matching clusters of the two partitions P_1 and P_2. The degree of similarity could be again computed by the Jaccard similarity coefficient. Adding these similarity coefficients for all possible cluster combinations, we come up with the following objective function.

$$f(P_1, P_2) = \sum_{A \in P_1} \sum_{B \in P_2} \omega_{A,B} * \mathrm{sim}(A, B)$$

To consider the different sizes of the cluster, we added a weighting coefficient ω. For two clusters A and B, the coefficient just sums up the number of elements of both clusters: $\omega_{A,B} := |A| + |B|$. It is independent of the similarity of the clusters and gives larger cluster combinations a higher weight.

4.1.2 Significance Level Thresholds

This objective function, however, only evaluates the level of detail with respect to conformance and does not consider significance. But it is difficult to balance significance and conformance in a single objective function. An appropriate balance might also depend on the concrete application the user has in mind.

To allow high conformance on different levels of significance, we introduce a significance level threshold for each of the two decompositions. This threshold prevents collapsing clusters beyond this level while optimizing the conformance.

The grid pattern in the upper left corner of the visualization displays the two significance level thresholds. The user is able to set both levels with a single click. For instance, in Figure 2 the user has clicked on the grid element at the intersection of the third column and the third row, indicated by a black box. This means that both decompositions have to stay expanded up to the third level while optimizing the conformance of the decompositions.

4.1.3 Optimization Algorithm

The two decompositions, the objective function, and the two significance level thresholds form a constrained maximization problem. As an optimization strategy for this problem, an exhaustive search, however, is not applicable for nontrivial data sets. The number of possible partitions induced by a single hierarchy might already grow exponentially with the number of leaf nodes n: In the worst case—a binary hierarchy—at least the $\frac{n}{4}$ intermediate nodes of the lowest collapsable level can be independently switched. This leads to at least $2^{\frac{n}{4}}$ different partitions.

Instead, we use a hill climbing algorithm to find a local maximum of the optimization problem. As an initialization, the algorithm expands the two decompositions to

the minimal level, which is defined by the two significance level thresholds. Then it tries to maximize the objective function as follows (expand operations that improve the objective function persist while all other expand operations are directly undone):

1. Expand each collapsed node of the first hierarchy one by one.
2. Repeat step (1) for all collapsed nodes of the second hierarchy.
3. If nothing has improved in step 1 and 2, try to expand two nodes concurrently, one in the first and one in the second hierarchy (systematically over the quadratic number of all possible combinations).

These three steps are repeated until they cannot provide any further improvement. The third step turned out to be helpful to skip local maxima because some pairs of clusters already match on a higher level, but even better on a lower level: Only expanding one of the clusters does not lead to a better match, but concurrently expanding both would do.

Thus, our optimization strategy provides an interactively selectable level of detail with an adaptive refinement to underline matching parts of the two decompositions. Clicking on a grid element of the threshold visualization (the grid pattern in the upper left corner of the diagram), the algorithm automatically produces a layout of few black boxes surrounded by many white ones. Figure 4 illustrates this process: While the image on the left hand side shows two totally expanded hierarchies, the image on the right hand side is actually created applying the optimization algorithm (this image is also depicted in Figure 2 in larger size).

4.1.4 Evaluation

Our impression using the visualization was that the adaptive level of detail algorithm drastically reduces the time to find an appropriate level of detail. In a brief evaluation we want to substantiate that the algorithm finds useful solutions and reduces the number of necessary interactions. We employ the real-world data set that is later also used in the case study (Section 5). The data set consists of six software projects. To test the algorithm, we try to find a matching level of detail between the package structure and a clustered decomposition of the systems created with the clustering tool Bunch.

We first identify an appropriately matching level by manually collapsing and expanding packages and clusters without using the adaptive algorithm. In particular, guided by the similarity metric that is encoded in the background of the matrix, we try to find a matching cluster for each of the major packages. When the solution satisfies our subjective assessment, we save the result and try to retrieve it by only using

1. the collapse and expand mechanism for single clusters,
2. the level collapse mechanism together with the collapse and expand mechanism for single clusters, or
3. the adaptive level of detail algorithm (Section 4.1.3) together with the collapse and expand mechanism for single clusters.

For each condition, we compute the optimal solution starting at the totally expanded hierarchy. The conditions are assessed by measuring the minimal number of interactions the user would need to retrieve the result. We repeat this procedure for every of the six software projects.

Results: On average, the users would at least have to perform 36.7 interactions in condition (1), 16.0 in condition (2), and 7.3 in condition (3) to retrieve the manually derived level of detail. This means that the algorithm in (3) dramatically decreases the number of necessary interactions: The user would have to use only 20% of the interactions required when exclusively using the standard collapse and expand interactions (1) or only 46% required when also including the level collapse mechanism (2). Moreover, the low number of interactions in (3) shows that the algorithm produces results close to the meaningful manual solution.

4.2 Sorting

The linear ordering of the rows and columns is elementary for the readability of a matrix graph visualization.[16] With a random ordering no structure would be visible, whereas a good ordering would reveal important graph structures like clusters, hub vertices, or outliers. Different approaches and algorithms exist to create a reasonable layout—Mueller et al.[16] as well as Henry and Fekete[17] survey these techniques in detail.

The hierarchical representation of the two software decompositions, however, constrains and partially defines this ordering in our visualization. If the decomposition follows a certain semantic, this mandatory sorting may already help revealing the structure of the dependency graphs. Nevertheless, the ordering still leaves some degree of freedom: The positions of sibling clusters and code entities can be switched without violating the constraints.

4.2.1 Local Diagonals

Interpreting matrix diagrams, the diagonal is an important reference line:[18] In a typical matrix representation of a graph, the cells on the diagonal represent self-edges. Our visualization depicts two decompositions at the same time. Thus, in contrast to most matrix graph visualizations, it uses different vertical and horizontal entity orders. A side effect is that the former diagonal entries, which we call *self-referencing cells* in the following, are scattered all over the diagram. To regain a local diagonal structure, the tool sorts sibling code entities without destroying the hierarchical structure.

If all self-referencing cells in a matrix are on the diagonal, an imaginary link between an arbitrary pair of self-referencing cells always forms a descending line (from left to right). This condition only holds globally for a perfect diagonal and thus usually cannot be established in our case due to the two different hierarchies. But we are able to fulfill the condition locally for each of the clusters on the lowest level of the hierarchies. We implemented an algorithm that aims at eliminating all ascending lines in those local clusters and thereby creating local diagonal structures. Figure 5 illustrates the algorithm: In the first decomposition, every pair of code entities in the same cluster

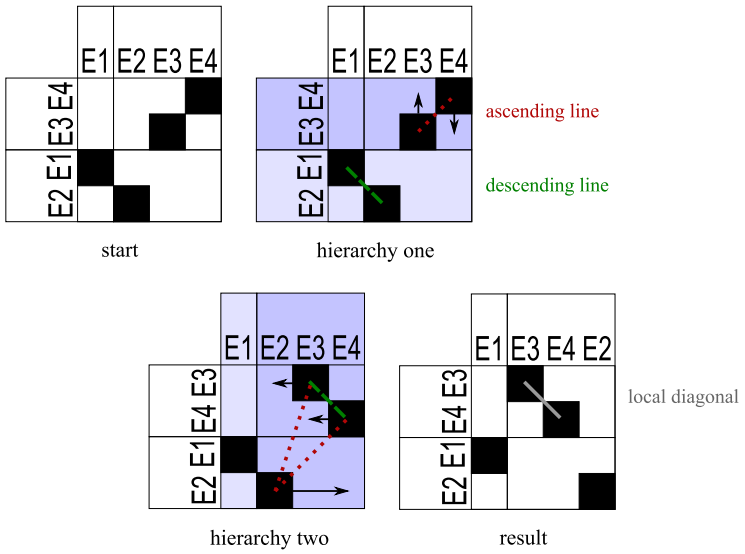


Figure 5: Sorting algorithm on leaf level. Black boxes mark self-referencing cells; ascending lines are dotted; descending lines are dashed; arrows indicate the transformations.

is switched if the two self-referencing cells define an ascending line. The same procedure is applied to the second decomposition. Finally, all lines between sibling elements are descending and form local diagonals.

Besides the locally regained diagonal structure, results of this local sorting algorithm are blocks of neighboring entities in the one decomposition that all belong to the same cluster in the other decomposition. These *mutual blocks* reveal additional important information for comparing the two decompositions: They show how a cluster in one decomposition is spread over the other decomposition.

The mutual blocks are encoded as boxes in the entity bars of the two decompositions. They look like a bar code and form the border lines between the icicle plots and the adjacency matrix (Figure 2). Each box represents a mutual block, which relates two clusters from the two decompositions. The brightness of the box corresponds to the Jaccard similarity of the two associated clusters: A black box stands for a good match while a gray or white box represents mutual blocks that only partially cover the two clusters.

Although the similarity of clusters is already encoded in the matrix background, these mutual blocks help detecting further interesting phenomena. Comparing the two forms of encoding, we observe that both are important because they support different tasks:

Matrix Background From the background encoding of the similarity metric, users are able to retrieve which cluster in one hierarchy is most similar to which cluster in

the other hierarchy. The background color gives a rough impression of the extent of similarity.

Mutual Blocks The mutual blocks are a kind of summary of the similarity information with respect to one hierarchy. For the particular hierarchy, they provide a better overview on which clusters are matched: The user only has to look at the hierarchy and its mutual blocks and does not need to search the whole matrix. Moreover, details can be retrieved in the mutual blocks with higher precision because the similarity is not only encoded in the color but also in the size of the blocks. For instance, we see at a glance that the *torrent* package in Figure 2 is only half matched. On the other hand, also small differences in mostly matching clusters become visible, as it is the case for the *disk* package in Figure 2. This ability is very important for analyzing evolving decomposition structures including only small changes.

4.2.2 Global Sorting

We usually cannot reconstruct the global diagonal because the two hierarchies constrain the ordering of vertices. Nevertheless, we may try to retrieve parts of it by placing the self-referencing cells as near as possible to the global diagonal. Analogously to local sorting, this goal can be expressed through minimizing the global number of ascending lines that connect the self-referencing cells.

We found that this optimization problem is equivalent to the problem of minimizing edge-crossings in a corresponding node-link based hierarchy comparison, which is called the *tanglegram layout problem*[19] or the *two-tree crossing minimization problem*,[20] an NP-hard problem. Figure 6 (top) provides an example of the corresponding visualizations for a graph consisting of five vertices. The diagram on the left shows our matrix-based comparison together with the ascending lines between the self-referencing cells, which measure the deviation from the global diagonal. The diagram in the middle depicts a transition step where the vertices in the matrix are connected by edges like in a node-link diagram—the number of edge crossings is equal to the number of ascending lines. Finally, the diagram on the right reflects the node-link based hierarchy comparison, which can be created by distorting the node-link diagram from the transition step. The distortion does not change the number of edge crossings.

This example already suggests the equivalence of the two problems, but the equivalence can also be proofed by induction as sketched in Figure 6 (bottom). We want to show that the number of ascending lines a in the matrix is equal to the number of edge crossings b in the transition step, is equal to the number of edge crossings c in the node-link comparison, i.e., $a = b = c$. The situation is trivial for $n = 1$ vertices, where neither ascending lines a nor edge crossings b, c exist. In the following we will show that, if the assumption holds for $n - 1$ vertices ($a_{n-1} = b_{n-1} = c_{n-1}$), it also holds for n vertices: Without loss of generality, we assume that the new vertex is added at the last position of the first hierarchy. Then, $0 \leq k \leq n - 1$ vertices are positioned after the corresponding new vertex in the second hierarchy. As Figure 6 shows, this means that the number of ascending lines increases by k because of k self-referencing cells in the respective block ($a_n = a_{n-1} + k$); in the transition step, the number of edge

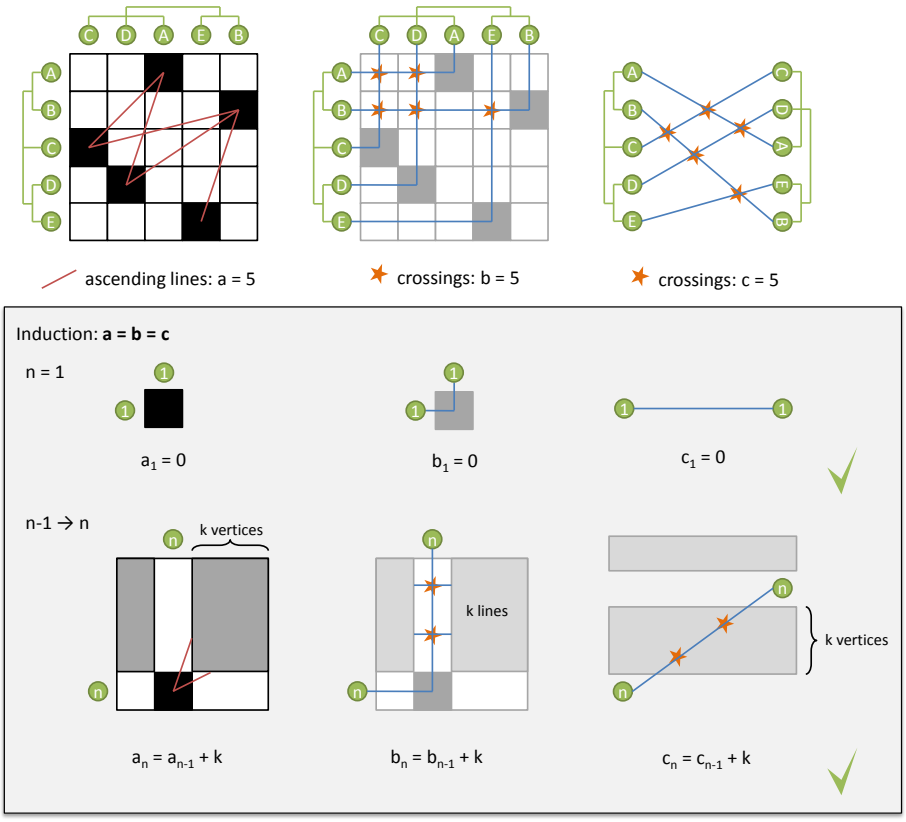


Figure 6: The equivalence of minimizing ascending lines between self-referencing cells in our matrix-based hierarchy comparison and minimizing edge crossings in a corresponding node-link diagram.

crossings increases by k because of k horizontal lines ($b_n = b_{n-1} + k$); and in the final node-link diagram, the number of edge crossings increases by k because of k skipped vertices ($c_n = c_{n-1} + k$). Hence, $a_n = b_n = c_n$.

Though the problem is NP-hard, there exist efficient heuristics. We implemented the hierarchy sort heuristic by Holten and van Wijk,[21] which is also able to handle non-binary hierarchies and runs in $O(n \cdot H)$, where n is the number of leave vertices and H is the maximum hierarchy height (and the number of collapse-and-expand cycles is constant).[19]

The algorithm is based on collapse and expand operations, which allow a level-by-level sorting of the hierarchies (as described in detail by Nöllenburg et al.[19]). A collapse-expand phase starts with the totally expanded hierarchies, optimizes the ordering of the leaf vertices for both hierarchies, and then collapses the two hierarchies to the next lower level. This process continues until the root vertex is reached and is then reverted by expanding the hierarchies again level by level while sorting the leaf vertices. The collapse-expand phases are repeated until no further improvements could be reached.

The optimization of leaf vertices is based on the barycentric method originally proposed by Sugiyama et al.[22] to lay out the levels of a hierarchical graph: The leaf vertices that are siblings in the hierarchy are ordered according to the barycenters of the set of vertices they are connected with in the other hierarchy. This is done for the first hierarchy while the second hierarchy is fixed and then repeated vice versa.

We perform global sorting once at start-up on the inner vertices of the completely expanded hierarchies. Since the procedure to retrieve the local diagonals is exact and simple, we still apply the local sorting algorithm to the code entities afterwards. Every time a hierarchy element is expanded or collapsed, local sorting is re-applied to always preserve local diagonals and mutual blocks. In contrast, global sorting does not need to be re-applied because expanding or collapsing a particular hierarchy element has only local effects and does not change the situation with respect to the other hierarchy elements.

4.2.3 Evaluation

To estimate the effect of sorting, Figure 7 provides four examples—(a) and (b) illustrate the effect of local sorting, (c) and (d) show the importance of global sorting.

Skipping local sorting as demonstrated in (a) has negative consequences for the mutual blocksin the entity bars of the hierarchies. The blocks are scattered among the leaves of the respective hierarchy element. Comparing (a) to (b) reveals that local sorting creates homogeneous mutual blocks, which help analyzing how the contained entities of a hierarchy element are distributed over the other hierarchy. Additionally, local sorting is necessary to reconstruct the local diagonals.

The effect of global sorting is that the self-referencing cells, which are distributed over the whole diagram without global sorting in (c), move closer to the diagonal of the matrix like shown in (d). The black and dark-gray rectangles, which contain many self-referencing cells, are thereby also aligned. This has some major advantages for comparing the two hierarchies: First, it is easier to get an overview on all matching clusters as we just have to follow the diagonal and do not have to search the whole

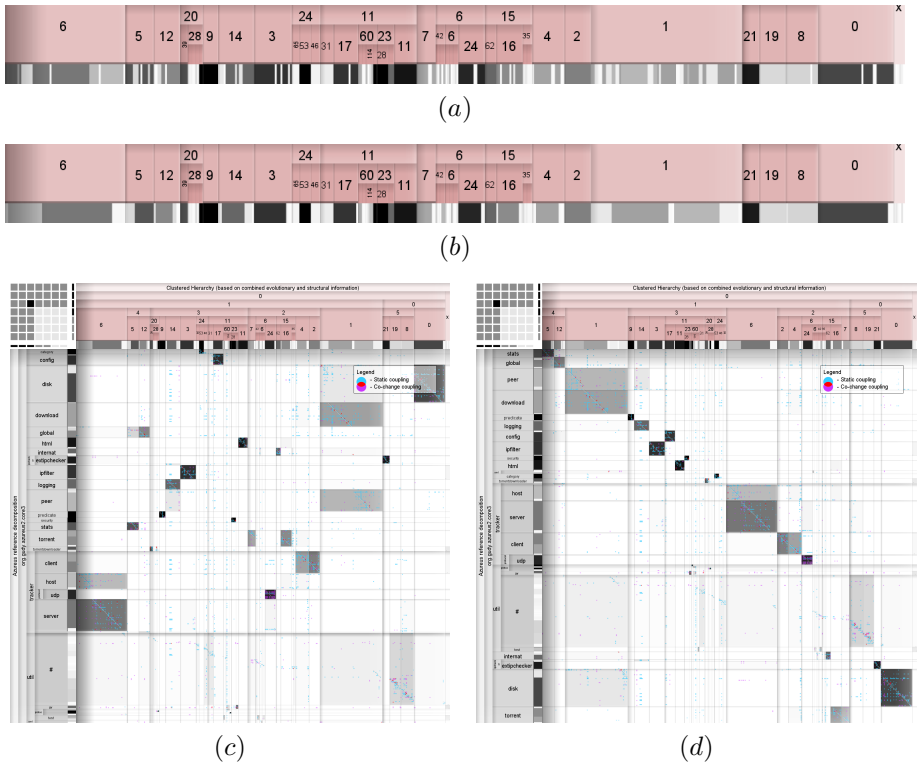


Figure 7: The effect of local sorting on mutual blocks: without sorting (a), with local sorting (b). The effect of global sorting on the background structure: without global sorting (c), with global sorting (d).

diagram as necessary in (c). Second, gaps in the diagonal easily reveal outliers. These outliers could be packages that are not well-matched and hence produce only light-gray boxes. Or we could find the missing dark box somewhere off the diagonal. This indicates that the two hierarchies indeed match at the current level of detail, but may conflict at a higher level because otherwise the algorithm would have been able to arrange the respective box near the diagonal. Third, the mid-gray boxes belonging to the same hierarchy element are placed near to each other. This reduces the effort to find the hierarchy elements that are united or split in the other hierarchy.

5 Case Study

The visualization approach was motivated by the application of studying software clustering results. Hence, the following visual analysis will apply our visualization technique to the previous study on software clustering.[6] The study incorporated the software clustering tool Bunch,[15] an approach based on the principle of high cohesion and low coupling of modules, to compare different data sources for software clustering. We assessed the clustered software decompositions retrieved from six sample projects by comparing them to a reference decomposition: the actual package structure of the project. As discussed in the Section 2, this quantitative assessment left some questions unanswered.

In the following we will analyze the software decompositions again, but now in a more qualitative and explorative approach. The tasks defined in Section 2.2 provide different views on the data sources and clustering results.

In general, our analyses consider all six sample projects of the study, namely, Azureus, JEdit, JFreeChart, JFtp, JUnit, and Tomcat. For practical reasons, we only depict the resulting visualizations for JFtp, the smallest of the sample projects, in the paper and provide respective visualizations of the other systems as supplementary material. Observations and findings that supplement the original analysis of the clustering results are reported in the following.

5.1 Compare Dependencies (Task 1)

The dependency graphs are the basis for the clustering process: They are the input for the clustering algorithm. On the one hand, we used static code dependencies—like inheritance, aggregation, and usage—to represent a traditional software clustering approach. These dependencies form the Structural Class Dependency Graph (SCDG). On the other hand, we used co-change couplings, which form the Evolutionary Class Dependency Graph (ECDG), to represent hidden dependencies. These co-change couplings relate two classes if these classes have been frequently changed together in the evolution of the software project. The dependency strength consists of a support value—the absolute number of co-changes—and a confidence value—a relative number of co-changes. To reduce the noise in the data set, a filter eliminates weak dependencies: We only consider two classes as coupled by co-change if the confidence value is higher than 0.8. This threshold value was derived from the empirical results in our previous study[6] where this setup tends to produce good clustering results. In this

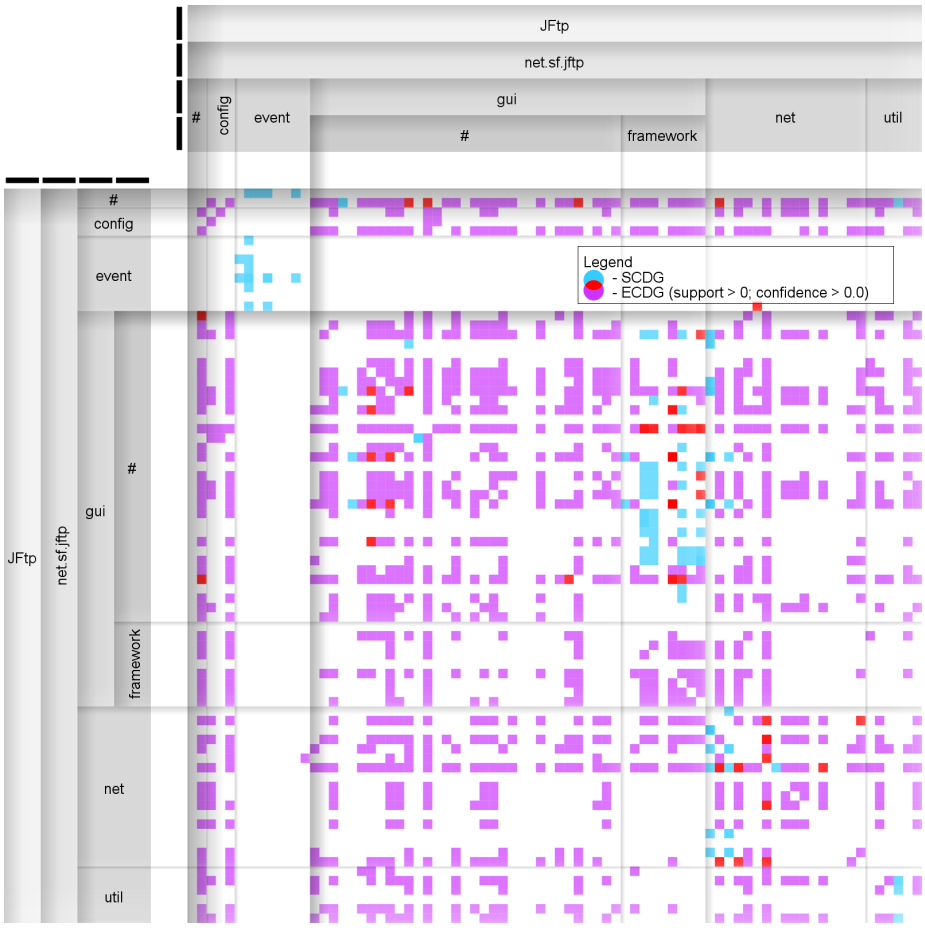


Figure 8: Graph comparison between the SCDG and the unfiltered ECDG for the JFtp project; the package structure provides a default decomposition.

first example unrelated to clustering, we wanted, however, to analyze the raw data and do not apply a filtering for co-change couplings.

The first phase of the case study uses the visualization as a graph comparison tool (Task 1). Since clustered decompositions are not yet relevant, the package structure is employed as default decomposition. The background structure thus does not carry any further information here. The graph visualization, however, reveals significant differences in the graph structures, as illustrated for JFtp in Figure 8:

SCDG (blue & red dependencies) Sparse graphs, but with dependencies that cover most of the nodes at least once. Some outstanding nodes with many incoming or outgoing dependencies form a kind of hub nodes.

ECDG (purple & red dependencies) Dense graphs (without filtering as shown in Figure 8) up to very sparse ones (with a strong filtering). Local concentrations of edges form dense clusters. But many nodes are not covered by any dependency.

These results show two main drawbacks of the ECDG: the local concentration of dependency information and the overall low density of the dependency graph, especially for stronger filtering setups.

Furthermore, the intersection of the dependencies (red dependencies) of both graphs is small and mostly relates classes of the same package. Since those dependencies that do not cross package borders help retrieving the package structure this tendency explains why it is beneficial to give those dependencies more weight in the clustering process.

5.2 Decompositions & Dependencies (Task 2)

In this second stage of our analysis, we also consider software decompositions produced by the employed clustering approach. We use the vertical axis to depict the clustered decomposition based on the structural code dependencies (SCDG) and the horizontal axis for the one based on the co-change dependencies (ECDG). Figure 9 shows such a visualization for the JFtp project.

In the evolutionary software decomposition (Figure 9, top), cluster *x* looks interesting: It roughly covers a third of the hierarchy, but is not subdivided further. There also exists a cluster *x* in the structural software decomposition (Figure 9, left), but it is much smaller. This cluster *x* represents all elements that could not be clustered because there was not any dependency information available for them. Thus, there are no co-change dependencies available for about a third of the classes of the software system, and the clustering algorithm could only cluster the other two thirds of the system. This situation is even worse in the other sample projects. This sparse coverage seems to be the main problem of clustering a software system exclusively with co-change information (ECDG).

Admittedly, we already uncovered this fact in the previous study. But there we needed a metric to measure the coverage, in contrast to the visualization, where we were able to grasp the same fact without even intentionally looking for it.

Analyzing the relation of the hierarchies and the graphs in more detail, we observe in the visualization that deeper hierarchies come along with clearly identifiable clusters indicated by visual clusters in the dependency graphs. The clustering algorithm seems to produce flatter hierarchies when the clusters in the dependency graphs are less clear. But we are not able to find any significant difference between structural and evolutionary decompositions with respect to this effect.

All in all, our case study confirms that the conformance between the structural and evolutionary decompositions is low. This might indicate that both data sources actually cover different dimensions of code dependencies—a combination of both data sources combines these two dimensions. Actually, this lead to slightly better clustering results as we found out in the previous study.

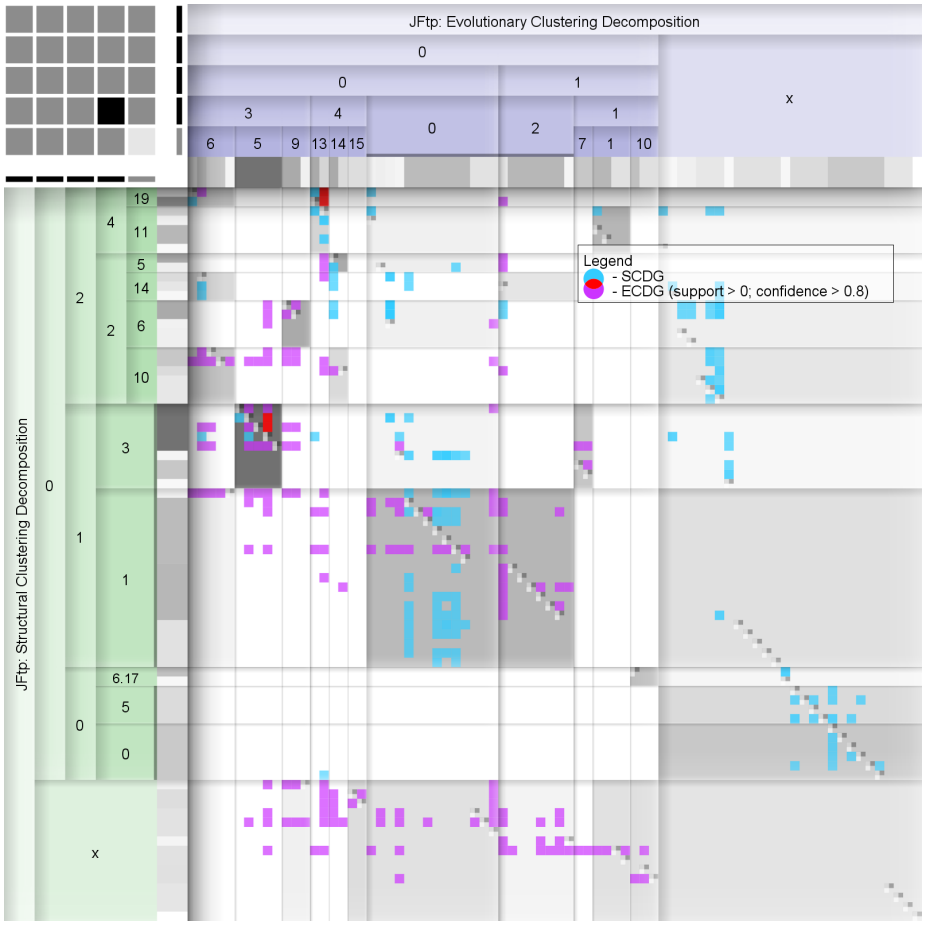


Figure 9: Two different clustered decompositions, one based on structural dependencies (vertical), the other based on co-change dependencies (horizontal).

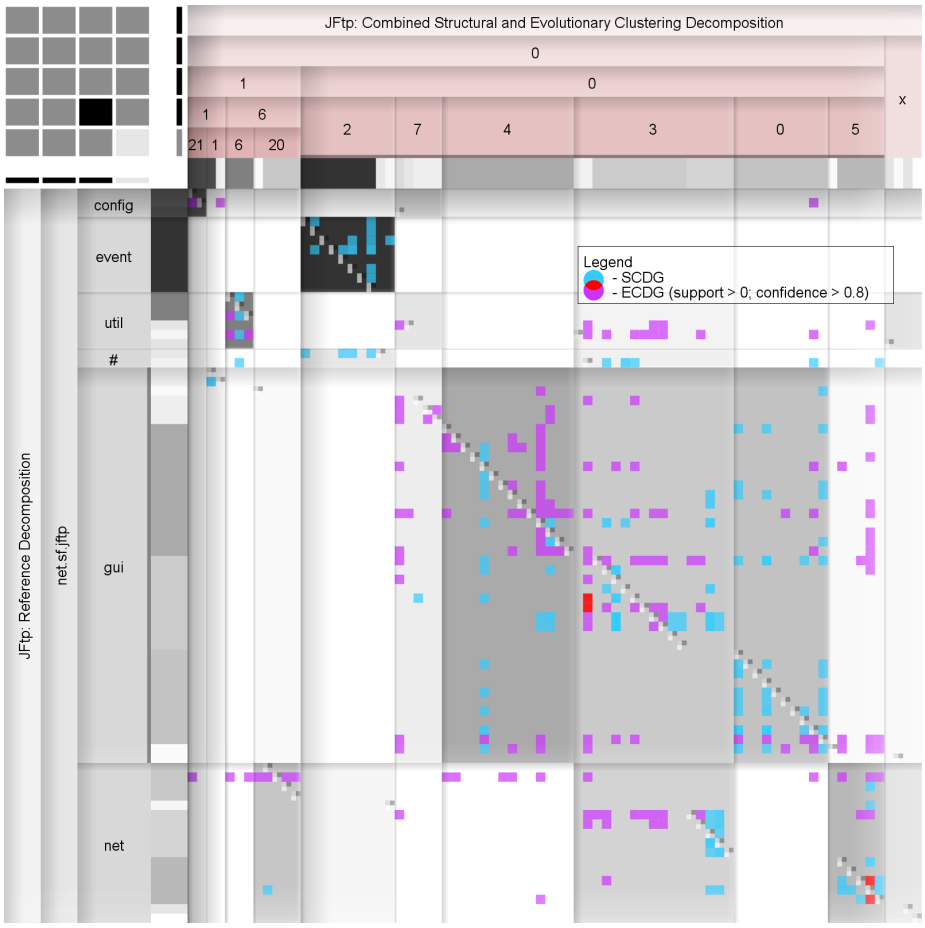


Figure 10: Clustered software decomposition based on the combined structural and evolutionary graphs compared with the reference decomposition (package structure).

5.3 Compare Decompositions (Task 3)

With our visualization the user is able to detect matching clusters at first glance—perhaps the most striking feature of the technique. For instance, in Figure 10 the *event* package is almost perfectly matched by cluster *0.0.2*, as we learn from the background shading of the matrix. The precondition is an appropriate level of detail, which could be easily gained by the level of detail optimization algorithm.

We use this ability of detecting well-matched packages to identify those packages that are either fairly matched or non-matched. In most cases, nearly perfectly matched packages possess a high structural cohesion, in other words, many structural dependencies connect the classes. Concurrently, those well-matched packages are sometimes, but less often supported by good co-change cohesion—the classes of the cluster were frequently changed together. In contrast, matching clusters predominantly based on

high co-change cohesion are rare. This explains why combining structural and co-change data improved the clustering quality in the quantitative study and using exclusively co-change data was only partly successful.

In contrast, utility packages—packages that provide some global functionality—could hardly be retrieved by our clustering approach. The visualization supports identifying those packages, even when they are not named *utility* or *util*, by their characteristic structure: Utility packages do not have outgoing dependencies to other non-utility packages, but many incoming ones from diverse packages. We are able to gain this information either by looking at the adjacency matrix or by using the interactive details-on-demand that highlight all adjacent classes (including the direction of adjacency) for a package as demonstrated in Figure 3. Once we identified these utility packages, the visually encoded cluster similarity revealed no significant correspondence to the clustered packages. This problem is a known problem of dependency-based clustering approaches.[23] A preprocessing that detects such packages before the actual clustering, like proposed by Mancoridis et al.,[15] might improve the clustering results.

Our visualization also showed that in some setups—in particular those involving the larger projects—the clustering algorithm was not able to create a decomposition with at least a roughly matching granularity: All possible levels were much too fine-grained in contrast to the reference decomposition. Repairing this weakness of the algorithm (e.g., by forcing the algorithms to produce more levels) might also result in much better clustering results for these setups.

5.4 Threats to Validity

The presented case study is a qualitative, task-oriented evaluation of the visualization tool, and as a consequence, does not quantitatively or comparatively measure the performance or effectiveness of the visualization. The case study was performed by the authors themselves and did not involve other users. For discussing the validity of this study in detail, we follow the criteria for judging qualitative research introduced by Lincoln and Guba.[24]

Credibility Using a data set derived from real-world software projects and analyzing the results of a published study on software clustering increases the credibility of the case study. On the other hand, the authors being the only participants of the case study limits its credibility.

Transferability By analyzing the results of the clustering experiment, the case study is focused on a specific example of comparing software architectures; in general, it cannot be assumed that the visualization technique works in other scenarios. Some aspects, however, suggest a certain transferability: Different software projects and tasks are analyzed, which shows a certain flexibility of the approach. The results include multiple kinds of observations, each kind having the potential to be transferable to other scenarios. Moreover, the data model is defined in detail, which enables the reader to rate the transferability at least from a technical point of view.

Dependability The case study is dependable as it could be replicated by reimplementing the visualization approach, which is described in sufficient detail, and applying it to a data set. When aiming at a close replication, the data set can be retrieved from open sources: the full process is described in previous work.[6] Nevertheless, interpreting visualizations has a strong subjective component—even a close replicate may come to different conclusions.

Confirmability Through describing the observations in detail and providing all visualizations as supplementary material, the results of the case study become confirmable. Sometimes, however, some additional knowledge on the original clustering experiment[6] could be necessary to validate the particular conclusions.

The case study shows, with some limitations of validity, how the visualization can be leveraged to interpret software clustering results in different tasks and software projects. It, however, does not allow direct conclusions on the applicability and effectiveness of the approach in a real-world software development scenario.

6 Related Work

Software architecture visualization is an established discipline in software visualization research.[4,25] Many tools from this area visualize software decompositions and code dependencies—*SHriMP*,[26] *Software Landscapes*,[27] or *Class Blueprints*,[28] just to name a few. Most of these visualizations employ the node-link metaphor to represent a dependency graph structure. But matrix-based visualizations of graphs seem to gain importance due to their advantages when it comes to visualizing larger graphs.[10] They have already been employed to analyze dependencies of software projects, for example, method calls[29] or co-change couplings.[30] Originating from the analysis of manufacturing processes, so-called *Dependency Structure Matrices* are also able to visualize software architectures in a matrix structure.[31] Due to a specialized sorting, these matrices help detecting cyclic dependencies and architecture violations. Recently, Zeckzer[32] presented an approach that is similar to ours as it aims at comparing different dependency types in software projects. Additional to using different colors, this approach splits each cell of the matrix into n pieces, whereby each piece represents a certain dependency type. This allows comparing more than two types of dependencies, but also reduces the scalability of the visualization.

Visualization has also played a role in software clustering and has helped to present single clustering decompositions in a readable way. Hierarchical decompositions have been depicted in a form of tree diagrams,[33] code dependencies have been represented as graph visualizations,[34] and similarity of code entities in high-dimensional feature spaces have been visualized as similarity matrices.[35] Other clustering related research communities use similar forms of cluster visualizations.

These visualizations are able to present a single software architecture description or a single clustering result. But to the best of our knowledge, no approach, however, uses a matrix visualization to concurrently compare different graphs and hierarchies, neither in the domain of software architecture visualization nor in clustering-related visualizations.

Nevertheless, there exist specialized visualizations to compare different hierarchies (without a graph structure). A straightforward approach is to place two hierarchies face to face with each other and connect related leaves by visual links. Edge crossings reduce the readability of such visualization. There exist heuristics that alleviate this problem by minimizing the number of crossings.[19] We employ one of these algorithms to globally order the entities of our matrix (Section 4.2.2). Holten and van Wijk[21] follow a different strategy and enhance the approach by bundling links into meaningful groups. Furthermore, brushing is another paradigm to express similarities of hierarchy nodes. For instance, TreeJuxtaposer[36] displays similar sub-tree structures interactively by highlighting the best corresponding node based on the Jaccard coefficient. Many other visualizations that compare hierarchical structures exist; Graham and Kennedy[37] provide a more exhaustive survey.

In the field of bioinformatics, *Cluster Heat Maps* are a popular visualization technique to analyze large clustered genome data sets.[38] These heat maps consist, first, of a color-coded matrix that usually relates genes (objects) to a set of conditions (attributes), and second, of an attached hierarchy retrieved by clustering. Not only the objects can be clustered, but also the attributes: A second hierarchy groups the attributes of the matrix. Concurrently finding an optimal clustering of objects and attributes is known as *biclustering* (e.g., Madeira and Oliveira[39] give an overview). These cluster heat maps, indeed, look similar to our approach, especially with two hierarchies attached. Nonetheless, the fundamental difference is that the cluster maps do not compare two hierarchies on the same set of objects, but help concurrently clustering two independent sets: objects and attributes.

Software clustering results are often evaluated by comparing them to a reference decomposition of approved quality. Like applied in our previous study a metric provides a similarity value. These metrics usually work on flattened decomposition. But there exist first approaches that additionally regard the hierarchical structure of the decompositions.[33,40] These metric-based approaches may be sufficient to get a quality measure for an automatically created decomposition, but do not explain the difference.

Other tools allow the user to visually compare graph structures. For instance, Andrews et al.[41] present a node-link approach to compare business processes and surveys related node-link approaches. Beside these specialized tools, every dynamic graph visualization approach enables graph comparisons: The two contrasted graphs form a sequence of changing graphs. There even exist dynamic compound graph visualizations, which are able to concurrently display a changing hierarchy.[42] These visualizations, however, are more suitable for evolving graphs and hierarchies, but not to contrast two totally different data sets like those discussed in this paper.

7 Applications in Software Visualization and Beyond

The introduced visualization technique was originally motivated by the need to compare software decompositions and different code dependency types in our research project. Hence, the primary users of the visualization are the authors themselves. The visualization provided additional insights as discussed in Section 5. Nevertheless, the approach might be applied to the following more general visualization problems. These

scenarios, however, represent only an outlook as they are not backed by the case study or other empirical evidence.

In the area of software visualization, the introduced technique might be directly used by software developers. The different graphs that a developer might want to compare could be call or aggregation graphs, inheritance dependency graphs, or other graphs consisting of code couplings like co-change or code clones. A project is usually decomposed by a dominating hierarchy, for instance, the directory or package structure of the system. Other hierarchies might not only stem from clustering, but could also be a different version of the package structure, a hierarchy induced by the cross-cutting concern in an aspect-oriented system, or the layers of the software architecture.

For instance, the growth of a software project might have led to a flawed architecture of a software system; the developers want to improve the design of the system by re-grouping the classes. They apply a clustering algorithm, but the proposed result is too far from their original design, which is a common problem when applying software clustering.[43] The visualization now helps comparing the original package structure and the clustering result; it links the hierarchy to the graphs by revealing the clusters that the algorithm detected. The background pattern shows non-matched packages, which could be candidates for re-structuring because the clustering algorithm was not able to confirm them. The algorithm might propose to split them or unite them with other packages. Later on, developers who did not take part in the re-modularization phase could track the changes by comparing the original decomposition of the system to the current one by also using the visualization technique.

The underlying data structure of the visualization—graphs and hierarchies—are not specific to software analysis. Basically, all hierarchical clustering approaches produce similar data: one or more hierarchies, which might be compared with each other or to a reference hierarchy, and one or more graph structures or similarity matrices that provide the clustering criterion. Hence, the visualization might be of interest in general for people who research on or apply clustering algorithms. Related to clustering, classification algorithms, another data mining technique, work with similar structures.

The three tasks that we derived from our application scenario (Section 2.2) are specializations of more general tasks. Abstracting the tasks allows looking for other areas of application, not limited only to software visualization and clustering research.

- Task 1 is based on the general task of *comparing graphs*.
- Task 2 is based on the general task of *analyzing compound graphs*.
- Task 3 is based on the general task of *comparing hierarchies*.

In summary, the three general tasks describe the problem of comparing compound graphs. Hence, the visualization could be applied in every application where compound graphs evolve or could be retrieved from different data sources. To give an example, when analyzing co-authorship in digital libraries, authors are connect by joint publications and hierarchically organized into communities. This compound graph structure changes over time, which can be analyzed using our visualization. Moreover, different community extraction algorithms may propose different results or competing hierarchical structures like the assignment to working groups, faculties, and universities could

be of interest. Furthermore, every application scenario of a hierarchy comparison[37] can be a potential application for the introduced visualization technique: In bioinformatics, different variants of phylogenetic trees classify species or, in ontology research, different hierarchical structures have to be mapped.

8 Conclusion

In this paper we analyzed how to compare software architectures with respect to software decompositions and code dependencies. To this end, we developed a novel visualization technique based on an adjacency matrix representation of graphs. The visual analysis of the results of a previous quantitative study on software clustering shows that the visualization supports the analysis tasks introduced in Section 2.2 in this scenario. The main capabilities of the visualization are

- concurrently contrasting software decompositions and code dependencies (Section 3.1 and Section 3.2),
- detecting matching and non-matching parts in software decompositions (Section 3.2), and
- semi-automatically finding a matching level of detail comparing two software decompositions (Section 4.1).

Our visualization technique is the first approach towards visually comparing architecture descriptions that consist of a decomposition of the software and code dependencies. While it has been a valuable research tool for our application, its effectiveness in other software engineering applications, however, is still to be verified (Section 5). In general, the visualization contrasts compound graph structures and might be of use in applications beyond software visualization like data mining, digital libraries research, bioinformatics, or ontology research (Section 7).

9 Funding

This work was supported by the Deutsche Forschungsgemeinschaft (DFG) [grant number DI 728/8-1/2].

References

[1] Bowman, I. T., Holt, R. C., and Brewster, N. V. (1999) Linux as a case study: its extracted software architecture. *ICSE '99: Proceedings of the 21st international conference on Software engineering*, New York, NY, USA, pp. 555–563, ACM.

[2] Maqbool, O. and Babri, H. A. (2007) Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering*, **33**, 759–780.

[3] Zimmermann, T., Diehl, S., and Zeller, A. (2003) How history justifies system architecture (or not). *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, Washington, DC, USA, IEEE Computer Society.

[4] Ghanam, Y. and Carpendale, S. (2008) A survey paper on software architecture visualization. Tech. rep.

[5] Beck, F. and Diehl, S. (2010) Visual comparison of software architectures. *SoftVis '10: Proceedings of the ACM 2010 Symposium on Software Visualization*, Salt Lake City, Utah, USA, pp. 183–192.

[6] Beck, F. and Diehl, S. (2010) Evaluating the impact of software evolution on software clustering. *WCRE '10: Proceedings of the 17th Working Conference on Reverse Engineering*, pp. 99–108, IEEE Computer Society.

[7] Wen, Z. and Tzerpos, V. (2004) An effectiveness measure for software clustering algorithms. *IWPC '04: Proceedings of the 12th International Workshop on Program Comprehension*, pp. 194–203, IEEE Computer Society.

[8] Chikofsky, E. J. and Cross, J. H. (1990) Reverse engineering and design recovery: a taxonomy. *Software, IEEE*, **7**, 13–17.

[9] Stevens, W. P., Myers, G. J., and Constantine, L. L. (1974) Structured design. *IBM Systems Journal*, **13**, 115–139.

[10] Ghoniem, M., Fekete, J. D., and Castagliola, P. (2004) A comparison of the readability of graphs using node-link and matrix-based representations. *INFOVIS '04: IEEE Symposium on Information Visualization*, pp. 17–24.

[11] Henry, N., Fekete, J. D., and Mcguffin, M. J. (2007) Nodetrix: a hybrid visualization of social networks. *IEEE Transactions on Visualization and Computer Graphics*, **13**, 1302–1309.

[12] Kruskal, J. B. and Landwehr, J. M. (1983) Icicle plots: Better displays for hierarchical clustering. *The American Statistician*, **37**, 162–168.

[13] Shneiderman, B. (1996) The eyes have it: A task by data type taxonomy for information visualizations. *VL '96: Proceedings of the 1996 IEEE Symposium on Visual Languages*, Washington, DC, USA, IEEE Computer Society.

[14] Rao, R. and Card, S. K. (1994) The table lens: merging graphical and symbolic representations in an interactive focus + context visualization for tabular information. *CHI '94: Proceedings of the SIGCHI conference on Human factors in computing systems*, New York, NY, USA, pp. 318–322, ACM.

[15] Mancoridis, S., Mitchell, B. S., Chen, Y., and Gansner, E. R. (1999) Bunch: A clustering tool for the recovery and maintenance of software system structures. *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, Washington, DC, USA, pp. 50–59, IEEE Computer Society.

[16] Mueller, C., Martin, B., and Lumsdaine, A. (2007) A comparison of vertex ordering algorithms for large graph visualization. *APVIS '07: Proceedings of the 6th International Asia-Pacific Symposium on Visualization*, pp. 141–148.

[17] Henry, N. and Fekete, J.-D. (2006) Matrixexplorer: a dual-representation system to explore social networks. *IEEE Transactions on Visualization and Computer Graphics*, **12**, 677–684.

[18] Mueller, C., Martin, B., and Lumsdaine, A. (2007) Interpreting large visual similarity matrices. *APVIS '07: Proceedings of the 6th International Asia-Pacific Symposium on Visualization*, pp. 149–152.

[19] Nöllenburg, M., Holten, D., Völker, M., and Wolff, A. (2008) Drawing binary tanglegrams: An experimental evaluation.

[20] Fernau, H., Kaufmann, M., and Poths, M. (2005) Comparing trees via crossing minimization. *FSTTCS '05: Foundations of Software Technology and Theoretical Computer Science*, vol. 3821 of *Lecture Notes in Computer Science*, pp. 457–469, Springer.

[21] Holten, D. and van Wijk, J. J. (2008) Visual comparison of hierarchically organized data. *Computer Graphics Forum*, **27**, 759–766.

[22] Sugiyama, K., Tagawa, S., and Toda, M. (1981) Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, **11**, 109–125.

[23] Andritsos, P. and Tzerpos, V. (2005) Information-theoretic software clustering. *IEEE Transactions on Software Engineering*, **31**, 150–165.

[24] Lincoln, Y. and Guba, E. (1985) *Naturalistic inquiry*. Sage focus editions, Sage Publications.

[25] Diehl, S. (2007) *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer.

[26] Storey, M. A., Best, C., and Michaud, J. (2001) SHriMP views: An interactive environment for exploring java programs. *IWPC '01: Proceedings ot the Ninth International Conference on Program Comprehension*, pp. 111–112.

[27] Balzer, M., Noack, A., Deussen, O., and Lewerentz, C. (2004) Software Landscapes: Visualizing the structure of large software systems. *VisSym '04: Proceedings of the Joint Eurographics - IEEE TCVG Symposium on Visualization*, pp. 261–266, Eurographics Association.

[28] Ducasse, S. and Lanza, M. (2005) The class blueprint: Visually supporting the understanding of classes. *IEEE Transactions on Software Engineering*, **31**, 75–90.

[29] van Ham, F. (2003) Using multilevel call matrices in large software projects. *INFOVIS '03: Proceedings of the IEEE Symposium on Information Visualization*, pp. 227–232.

[30] Burch, M., Diehl, S., and Weißgerber, P. (2005) Visual data mining in software archives. *SoftVis '05: Proceedings of the 2005 ACM Symposium on Software Visualization*, New York, NY, USA, pp. 37–46, ACM Press.

[31] Sangal, N., Jordan, E., Sinha, V., and Jackson, D. (2005) Using dependency models to manage complex software architecture. *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, New York, NY, USA, October, vol. 40, pp. 167–176, ACM.

[32] Zeckzer, D. (2010) Visualizing software entities using a matrix layout. *SOFTVIS '10: Proceedings of the 5th international symposium on Software visualization*, New York, NY, USA, pp. 207–208, ACM.

[33] Shtern, M. and Tzerpos, V. (2004) A framework for the comparison of nested software decompositions. *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, Washington, DC, USA, pp. 284–292, IEEE Computer Society.

[34] Mancoridis, S., Mitchell, B. S., Rorres, C., Chen, Y., and Gansner, E. R. (1998) Using automatic clustering to produce high-level system organizations of source code. *IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension*, Washington, DC, USA, pp. 45–52, IEEE Computer Society.

[35] Kuhn, A., Ducasse, S., and Gîrba, T. (2005) Enriching reverse engineering with semantic clustering. *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*, Washington, DC, USA, pp. 133–142, IEEE Computer Society.

[36] Munzner, T., Guimbretière, F., Tasiran, S., Zhang, L., and Zhou, Y. (2003) Treejuxtaposer: scalable tree comparison using focus+context with guaranteed visibility. *ACM Transactions on Graphics*, **22**, 453–462.

[37] Graham, M. and Kennedy, J. (2009) A survey of multiple tree visualisation. *Information Visualization*, **9**, 235–252.

[38] Wilkinson, L. and Friendly, M. (2009) The history of the cluster heat map. *The American Statistician*, **63**, 179–184.

[39] Madeira, S. C. and Oliveira, A. L. (2004) Biclustering algorithms for biological data analysis: A survey. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, **1**, 24–45.

[40] Shtern, M. and Tzerpos, V. (2007) Lossless comparison of nested software decompositions. *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*, Washington, DC, USA, pp. 249–258, IEEE Computer Society.

[41] Andrews, K., Wohlfahrt, M., and Wurzinger, G. (2009) Visual graph comparison. *IV '09: Proceedings of the 13th Conference on Information Visualisation*, pp. 62–67, IEEE Computer Society.

[42] Pohl, M. and Birke, P. (2008) Interactive exploration of large dynamic networks. *VISUAL '08: Proceedings of the 10th international conference on Visual Information Systems*, Berlin, Heidelberg, pp. 56–67, Springer-Verlag.

[43] Glorie, M., Zaidman, A., van Deursen, A., and Hofland, L. (2009) Splitting a large software repository for easing future software evolution - an industrial experience report. *Journal of Software Maintenance and Evolution: Research and Practice*, **21**, 113–141.