First publ. in: Proceedings / IEEE International Workshop on Visualizing Software for Understanding and Analysis (VisSoft), 2005, pp. 89-94

Exploring Relations within Software Systems Using Treemap Enhanced Hierarchical Graphs

Michael Balzer Oliver Deussen

Department of Computer and Information Science
University of Konstanz, Germany
{balzer | deussen}@inf.uni-konstanz.de

Abstract

The clear and meaningful visualization of relations between software entities is an invaluable tool for the comprehension, evaluation, and reengineering of the structure of existing software systems. This paper presents an interaction and representation scheme for the visualization and exploration of complex hierarchical graphs to analyze relations within software systems. Thereby aggregated parts of the software system are represented as treemaps that visualize the structure of the contained software entities. An adaption of existing rectangle-based treemap algorithms for layouts within convex polygonal bounding geometries is introduced to allow for a differentiation of various entity types in the graph visualization. Furthermore, a visual clustering method based on implicit surfaces is presented to create meaningful visualizations of distorted hierarchical graphs of software systems.

1 Introduction

Modern software systems are very complex structures consisting of thousands of entities and millions of lines of code. The entities are organized in a hierarchy that reflects the architecture of the software system. Typical hierarchy levels of software entities are nested subsystems, packages, modules, functions, classes, methods, and attributes, whereby in large systems one may find up to 20 or more of these levels. In particular, object-oriented software systems are constructed using an explicit and rich hierarchical structure provided by modelling and programming languages like UML and Java/C++.

Among the hierarchy of the entities, the relations between the entities provide important information about the structure of a software system. Examples for such relations are method calls, read or write accesses to attributes, and inheritance of classes. In contrast to the hierarchy that reflects the intended structure of the system, these relations provide insight into the inner dependencies of the entities as they arise during the development and reengineering process.

Software visualization can help to understand the structure of complex software systems. The approach of treemaps [10] is commonly used for visualizing hierarchical structures, and it is also applied to software structures [2, 13]. Another common method in the field of software visualization are graphs [11, 15, 17]. They are especially used for the representation of relations and dependencies within software structures. Additionally, they may be extended to hierarchical graphs [8, 12], thereby allowing the visualization of the hierarchical structure simultaneously.

For the comprehension, evaluation, and reengineering of existing software systems, it is necessary to provide views on different levels of abstraction. For example, global views illustrating dependencies between packages, detailed views showing relations of specific methods or attributes, and mixed views presenting relations between some selected classes and other packages. In most of these views, parts of the data is aggregated according to the hierarchical structure. This focus and context approach is necessary to filter the often immense amount of presented information according to the user's demand. The challenging task is to represent the aggregated parts of the system without losing information of their contained structure and complexity.

This paper presents a technique that creates meaningful visualizations of the relations within hierarchical software structures by enhancing graphs with treemaps. Thereby each aggregated node in the graph is represented by a treemap that reflects the contained structure of the node that is hidden in the graph layout. By interactively collapsing and expanding the presented parts of the data, the user is able to stepwise explore the dependencies even within very complex software systems.

2 Background

2.1 Hierarchical Graphs

A *graph* $G = (V, E)$ consists of a finite set of nodes V and a finite set of edges E with $E \subseteq V \times V$. A *directed graph* is a graph whose edges possess a direction given by a source node s and a target node t with $t, s \in V$.

A *hierarchical graph* $H = (G, T)$ consists of a directed graph G and a rooted tree T, such that the nodes of G are a subset of the nodes of T. The tree T is called the hierarchy tree of H and the directed graph G is called the underlying graph of H. Figure 1 presents an example of a hierarchical graph where the nodes are represented as boxes, the black edges represent the directed graph G and the gray edges illustrate the rooted tree T.

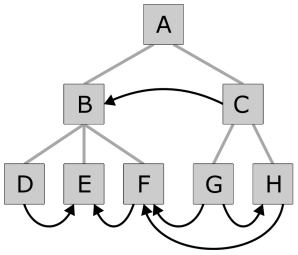

Figure 1. Hierarchical graph—black edges represent the directed graph and gray edges illustrate the rooted hierarchy tree

A *view* $G' = (V', E')$ of a hierarchical graph $H = (G, T)$ is a directed graph with a set of nodes $V' \subset V$ that contains exactly one ancestor of every node of the underlying directed graph $G = (V, E)$ of H according to the rooted hierarchy tree T of H. Figure 2 presents three examples for views of the hierarchical graph in Figure 1. In the first view all nodes are aggregated in node A. In the second view the nodes G and H are aggregated in node C. The third view presents all leaf nodes of the hierarchy tree.

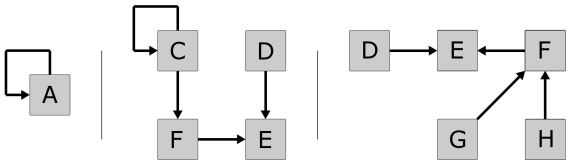

Figure 2. Three different views of the hierarchical graph in Figure 1

A *n-dimensional layout* of a directed graph $G = (V, E)$ is a vector of node positions (p_v) with $v \in V$ and $p_v \in \mathbb{R}^n$. An *energy model* formalizes what is considered a good layout by assigning an energy value to each layout of a given graph, whereby smaller energy values indicate better layouts [5, 7, 12]. For the computation of good layouts, which are based on a given energy model, there exist efficient iterative minimization algorithms [3, 14].

2.2 Treemaps

Treemaps [10] subdivide a given rectangular display area according to an attributed hierarchy without producing holes or overlappings. Thereby the term 'attributed' signifies that each node in the hierarchy has a value that represents its size relating to a given measurement.

The construction of treemaps is exemplified in Figure 3. Each node in the hierarchy has a name and an associated size, whereby the size of an internal node is the sum of the sizes of its contained leaf nodes. The treemap is constructed via recursive subdivision of the initial rectangle. The direction of each one-dimensional subdivision step alternates per level: first horizontally, next vertically, again horizontally, etc. The area size of each sub-rectangle corresponds to the size of the represented node. As a result of its construction, the treemap reflects the structure of the tree and the sizes of its nodes.

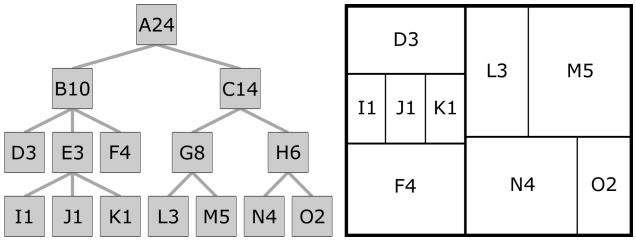

Figure 3. Tree and corresponding Treemap—each node is labeled with its name and size; the area sizes in the Treemap correspond to the node sizes

Beside this initial Slice-and-Dice method, more sophisticated treemap layout algorithms emerged, which mainly address the issue of the bad aspect ratio between width and height of the rectangles in the treemap by performing a two-dimensional subdivision in each recursion step. The popular Squarified treemap algorithm [6] sorts the nodes at each hierarchy level by their sizes in descending order and generates layouts with an aspect ratio converging to one. Ordered treemap [16] layouts preserve a given order of elements by also maintaining a good overall aspect ratio. In contrast to all other existing treemap layout algorithms, which are based on rectangles, Voronoi treemaps [1] are based on the subdivision of, and into polygons. They maintain a good aspect ratio of the elements, provide a non-ambiguous interpretability of the hierarchical structure, and enable treemap layouts within arbitrary polygons.

3 Contribution

The clear and meaningful visualization of relations between software entities is an invaluable tool for the comprehension, evaluation, and reengineering of the structure of existing software systems. A common method for this task is the usage of directed graph visualizations. Therefore, the software entities are represented as nodes of the graph, and the relations between these software entities are represented as edges between the according nodes.

Modern software systems are very complex and consist of thousands of entities and millions of lines of code. If the graph visualization presents the complete graph, the user is swamped and is not able to extract the information he/she is looking for. Thus, the presented data has to be filtered to views of the complete graph, whereby the user should decide what information is essential. Therefore, the hierarchical structure of the data can be exploited by aggregating parts of the software system to entities of higher levels in the hierarchy. The challenging task is to represent these aggregated parts without losing information of their contained structure and complexity.

In the following Section 3.1 we present the interaction scheme for handling complex hierarchical graphs, and introduce an approach for the visual representation of aggregated parts of the graph based on treemaps. In Section 3.2 we describe the graph layout method that is used for our visualization, including their properties. In Section 3.3 we introduce an adaption of conventional rectangular treemap layout algorithms for handling convex polygonal bounding geometries. In Section 3.4 we present an extension of the visual representation of hierarchical graphs using implicit surfaces. Results for software visualizations created with our techniques are shown in Section 3.5.

3.1 An Interaction and Representation Scheme for Hierarchical Graphs

The starting point for the user of our visualization of the software system as a hierarchical graph, is the representation of the complete system solely by the root node of the graph as its simplest view. In the next step the user expands this view by expanding the root node into its child nodes at the first hierarchy level that are related to the child entities of the root entity in the software system. The user analyzes the relations between the entities and further explores the dependencies in other views by expanding the nodes that represent the entities he/she is interested in. Complementary to the expansion step, it is also possible to collapse parts of the presented directed graph to entities at higher hierarchy levels. With this interaction scheme between different views of the same graph, the user is able to reduce the amount of presented information with regard to his analysis task, and to stepwise explore the relations within the software system.

To provide an insight into the aggregated parts of the software system that are each represented just by a single node in the view of the graph, these graph nodes are not only represented as simple geometric shapes, but rather visualized as treemaps. Each treemap representation of a graph node is generated according to the hierarchical structure of the represented aggregated part of the software system. The size of an entity in the treemap is related to the number of its connected relations. This enables the illustration of the internal structure of the aggregated parts of the software system, and the identification of the degree of connectivity of each aggregated entity simultaneously. For our treemap layouts we use the approach of Squarified treemaps [6], which sorts the nodes in the treemap by their sizes allowing a fast detection of entities with a high degree of connectivity.

Figure 4 exemplifies this interaction and representation scheme by an elementary data set. The first view shows a single treemap within a quad that represents the root package of the software system and reveals the hierarchical structure of all contained packages and classes. Large entities in the treemap indicate a high degree of connectivity. In the next step this root package is expanded to the contained two child packages, which each again visualizes its hierarchical structure of the contained entities by a treemap. In steps three and four these two packages are expanded as well, finally revealing all classes of the software system that are represented by circles.

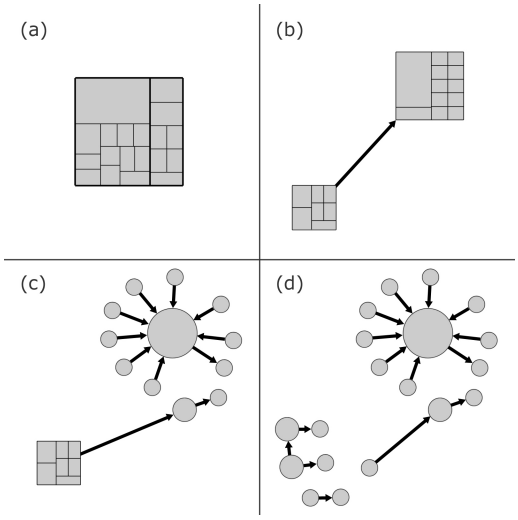

Figure 4. Stepwise exploration of a hierarchical graph—aggregated nodes are represented as treemaps illustrating their contained structure; the size of the entities indicate their degree of connectivity

3.2 Hierarchical Graph Layouts

The essential requirement for graph-based software visualizations is the generation of meaningful graph layouts. Existing visualizations present two-dimensional as well as three-dimensional graph layouts. The results of empirical studies that compare the effectiveness of two-dimensional and three-dimensional graph layouts are mixed: In some studies, 3D visualizations outperformed 2D visualizations [18], other studies yielded the opposite result [19]. In our experience with 3D layouts of large graphs, individual objects are often occluded and therefore barely recognizable, also orientation is often intricate. For this reason, we are using two-dimensional graph layouts to avoid visual clutter and to better preserve the user's mental map.

The quality of graph layouts is directly related to the properties of the used energy model and its appropriateness for a given visualization task. For our visualization we adapted the energy model of Noack and Lewerentz [12] that was developed particularly for software visualization. This model has several advantages: Firstly, it offers three parameters (clustering, hierarchicalness, distortion) to influence the characteristics of the layouts, whereby each parameter has a clear interpretation. Especially the variable degree of hierarchicalness allows to directly influence the importance of the hierarchical structure versus the importance of the relations in the generated layouts. Secondly, the node sizes in the graph layout directly correspond to the number of connected edges of or within each node. Thereby these node sizes are not independent of the graph layout, but they are rather directly related to a repulsion force for each node, which results in energy minimized layouts without any overlapping of the nodes. Thirdly, clusters are clearly separated in the energy minimized layouts.

To generate layouts with minimized energy values, we use an enhancement of the Barnes-Hut algorithm [3] by Quigley and Eades [14]. The combination of the energy model and this minimization algorithm allows us to generate our layouts at interactive rates.

3.3 Rectangular Treemap Layouts for Convex Polygonal Bounding Shapes

Software systems contain different types of entities, and relations often occur between these different entity types. Thus, the graph nodes have to be represented differently for each entity type. Here it is most appropriate to use well-defined geometric shapes. These shapes are also the bounding geometries for our treemap representations of the aggregated nodes, which implies that we need a method for generating treemap layouts within diverse shapes. One possibility is to use Voronoi treemaps [2], which enable treemap layouts within arbitrary polygons. However, this method is not appropriate for our interactive visualization task because of its time expensive computation. Instead we adapt the existing rectangle-based treemap layouts to a larger set of bounding geometries.

The principle mechanism of all rectangle-based treemap algorithms is to split a given area into two subareas by a straight horizontal or vertical line according to a given ratio between the subarea sizes. To achieve this, we have to restrict the bounding geometries to convex polygons, since the split of non-convex polygons may result in more than two subareas. This restriction is acceptable as the set of convex polygons offers a sufficient number of distinguishable shapes.

The horizontal or vertical split of rectangles is trivial. For the split of arbitrary convex polygons, we first have to find the trapezoid in which the split occurs. We use a scan line parallel to the splitting direction that allows us to test for each vertex in the polygon whether the split occurs before or after the current scan line position. If the split has to occur before the current position, we determine the trapezoid formed by the scan lines at the current and the foregoing position, and the edge segments of the initial polygon that are positioned between these two scan line positions. Then the exact split of this trapezoid is calculated. This split algorithm is applicable to all treemap layout algorithms based on rectangles.

This procedure is further illustrated by Figure 5: The given polygon $ABCDE$ has to be horizontally subdivided into two polygons p_1 and p_2 according to their desired area sizes. The two vertices between which the split occurs have to be found. These two vertices B and D and the corresponding two vertices at the opposite side of the polygon H_1 and H_2 form a trapezoid. The split points S_1 and S_2 are computed within this trapezoid, which is mathematically trivial.

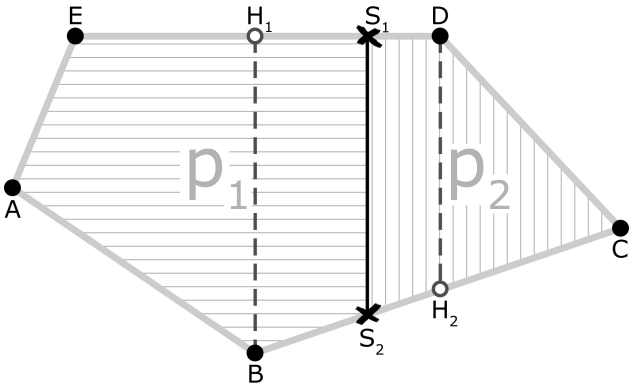

Figure 5. Example for a horizontal split operation of a convex polygon used for the subdivision step in treemap layouts

3.4 Implicit Surfaces for Visual Graph Clustering

The representation of the hierarchical structure of nodes in a hierarchical graph is rather difficult, because of the distortion originated by the relations between different parts of the hierarchy. An obvious and appropriate method for our software visualization is to color each node by its membership in a package, which means that each package and its contained entities are assigned to an individual color. Nevertheless, it is difficult to locate all entities that are contained in a specific package due to the distortion of the graph. To further support the user, we additionally implemented a visual clustering technique based on implicit surfaces [4] following the approach presented in [9]. We generate a two-dimensional implicit surface for each package in the graph that encloses all nodes that represent entities of this package with a color similar to the color of the package. The advantage of this implicit surface technique in comparison with the usage of simple boxes or circles, is that the areas highly adapt their shape to the distribution of the nodes resulting in an individual characteristic shape for each cluster. These shapes may also be made up of two or three independent areas for each cluster, which is a meaningful indication of the disruption of the according package.

3.5 Results

Finally, we present three exemplary results of our visualization: Figure 6 presents the complete call graph between classes of two main packages of the software system 'JFree'. In Figure 7 the involved entities of unwanted write accesses of attributes between two packages in 'JFree' are worked out, whereby uninvolved entities are aggregated. Figure 8 visualizes the inheritance relations between the top level packages of the 'Java Development Kit 1.4.2'.

Nodes in the graph visualization that represent packages are drawn as quads, classes as circles, methods as triangles, and attributes as diamonds. The color of a node represents its package membership. The direction of an edge is indicated by a gradient from dark to bright. The width of an edges indicates the number of represented relations.

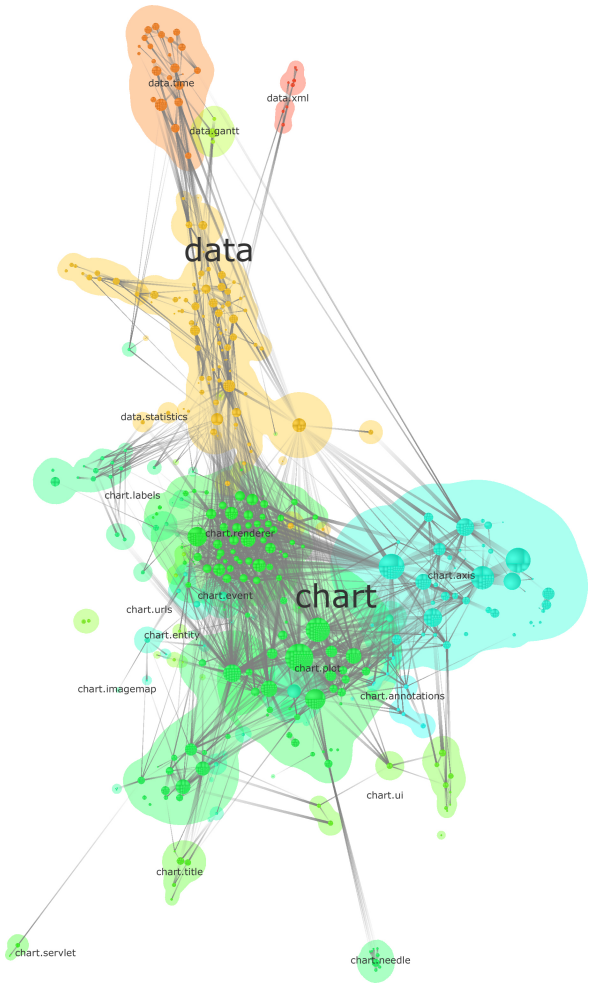

Figure 6. Call graph between classes of two main packages of the software system 'JFree'

4 Conclusion

We presented a new interaction and representation scheme for the visualization and exploration of complex hierarchical graphs to analyze relations within software systems. Thereby aggregated parts of the software system are represented as treemaps that visualize the structure of the contained software entities. To adapt existing rectangle-based treemap layout algorithms to convex polygonal bounding geometries, we introduced an alternative method for the horizontal and vertical subdivision of convex polygons. This method allows the usage of differently shaped treemap bounding geometries in order to differentiate between the entity types in the hierarchical graph. Furthermore, we implemented a visual clustering based on implicit surfaces to create meaningful visualizations of distorted hierarchical graphs.

The presented techniques enable the stepwise exploration of relationships such as inheritances, method calls or attribute accesses even within complex software systems. The user driven generation of views of a software system allows to work out single dependencies of entities or subsystems of the software as well as to create overviews of the relations within the complete software system or its parts. With these views, it is possible to validate the architecture, to identify problem areas, and to analyze reengineering alternatives.

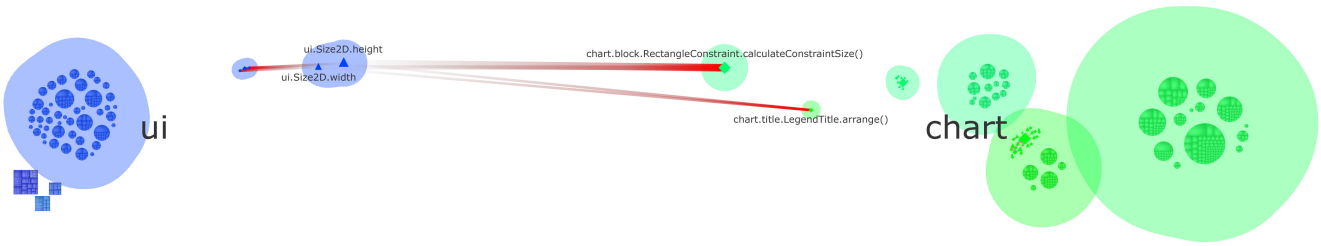

Figure 7. Entites that are involved in unwanted write accesses between two packages in 'JFree'

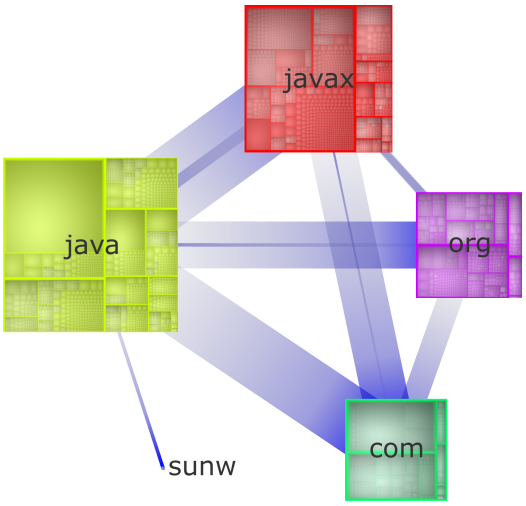

Figure 8. Inheritance relations between the top level packages of the 'JDK 1.4.2'

References

[1] M. Balzer and O. Deussen. Voronoi treemaps. In *Proceedings of the IEEE Symposium on Information Visualization*. IEEE Computer Society, 2005.

[2] M. Balzer, O. Deussen, and C. Lewerentz. Voronoi treemaps for the visualization of software metrics. In *Proceedings of the 2005 ACM symposium on Software Visualization*, pages 165–172. ACM Press, 2005.

[3] J. Barnes and P. Hut. A hierarchical O(NlogN) force-calculation algorithm. *Nature*, 324(4):446–449, Dec. 1986.

[4] J. F. Blinn. A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, 1(3):235–256, 1982.

[5] U. Brandes. Drawing on physical analogies. In M. Kaufmann and D. Wagner, editors, *Drawing Graphs: Methods and Models*, pages 71–86. Springer-Verlag, Berlin, 2001.

[6] M. Bruls, K. Huizing, and J. J. van Wijk. Squarified treemaps. In *Proceedings of the Joint Eurographics and IEEE TCVG Symposium on Visualization*, pages 33–42. IEEE Computer Society, 2000.

[7] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[8] P. J. Finnigan, R. C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. A. Müller, J. Mylopoulos, S. G. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, 1997.

[9] M. Gross, T. C. Sprenger, and J. Finger. Visualizing information on a sphere. In *Proceedings of IEEE Information Visualization '97*, pages 11–16. IEEE Computer Society, 1997.

[10] B. Johnson and B. Shneiderman. Tree maps: A space-filling approach to the visualization of hierarchical information structures. In *Proceedings of the 2nd International IEEE Visualization Conference*, pages 284–291. IEEE Computer Society, 1991.

[11] C. Lewerentz and A. Noack. CrocoCosmos – 3d visualization of large object-oriented programs. In M. Jünger and P. Mutzel, editors, *Graph Drawing Software*, pages 279–297. Springer-Verlag, 2003.

[12] A. Noack and C. Lewerentz. A space of layout styles for hierarchical graph models of software systems. In *Proceedings of the 2005 ACM symposium on Software Visualization*, pages 155–164. ACM Press, 2005.

[13] A. Orso, J. A. Jones, and M. J. Harrold. Visualization of program-execution data for deployed software. In *Proceedings ACM 2003 Symposium on Software Visualization*, pages 67–76. ACM Press, 2003.

[14] A. J. Quigley and P. Eades. FADE: Graph drawing, clustering and visual abstraction. In *Proceedings of The Eighth International Symposium on Graph Drawing*, pages 197–210. Springer-Verlag, 2000.

[15] S. P. Reiss. An engine for the 3d visualization of program information. *Journal of Visual Languages and Computing*, 6(3):299–323, 1995.

[16] B. Shneiderman and M. Wattenberg. Ordered treemap layouts. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 73–78. IEEE Computer Society, 2001.

[17] M.-A. Storey and H. A. Müller. Manipulating and documenting software structures using SHriMP views. In *Proceedings of the International Conference on Software Maintenance*, pages 275–284. IEEE Computer Society, 1995.

[18] C. Ware and G. Franck. Evaluating stereo and motion cues for visualizing information nets in three dimensions. *ACM Transactions on Graphics*, 15(2):121–140, 1996.

[19] U. Wiss and D. A. Carr. An empirical study of task support in 3d information visualizations. In *Proceedings of the International Conference on Information Visualisation (IV)*, pages 392–399. IEEE Computer Society, 1999.