Software Visualization via Hierarchic Micro/Macro Layouts

Martin Nöllenburg[1], Ignaz Rutter[2] and Alfred Schuhmacher[2]
[1]*TU Wien, Vienna, Austria*
[2]*Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany*

Keywords: Hierarchical Graph Layout, Compound Graphs, Software Visualization.

Abstract: We propose a system for visualizing the structure of software in a single drawing. In contrast to previous work we consider both the dependencies between different entities of the software and the hierarchy imposed by the nesting of classes and packages. To achieve this, we generalize the concept of micro/macro layouts introduced by Brandes and Baur (Baur and Brandes, 2008) to graphs that have more than two hierarchy levels. All entities of the software (e.g., attributes, methods, classes, packages) are represented as disk-shaped regions of the plane. The hierarchy is expressed by containment, all other relations, e.g., inheritance, functions calls and data access, are expressed by directed edges. As in the micro/macro layouts of Brandes and Baur, edges that "traverse" the hierarchy are routed together in channels to enhance the clarity of the layout. The resulting drawings provide an overview of the coarse structure of the software as well as detailed information about individual components.

1 INTRODUCTION

Source code is the natural textual representation of software. While it can be read and modified by humans and is suitable for transformation into an executable, it lacks communicative power on larger scales. Beyond the smallest software, getting familiar with a software by means of the code alone or even analyzing the overall condition of a software based on only its code is a difficult if not impossible task. It is therefore desirable to create visualizations of software that give a better overview, facilitate exploration of a software and allow to recognize the dependencies of certain parts of a software.

In this paper we propose and demonstrate a graph-based, node-link diagram visualization technique for software extending the micro/macro layout style of Baur and Brandes (Baur and Brandes, 2008) to multiple hierarchy levels. It is natural to model software by a graph that contains a vertex for each entity in the source code, such as packages, classes, methods, and fields. Relations are modeled as (usually directed) edges between these entities; for example code couplings such as inheritance, method calls, and field accesses. One characteristic of graphs obtained from software is that they usually contain a very strong multi-level hierarchy in addition to non-hierarchic relations; methods and fields are contained in classes, which are contained in packages, which may again be grouped in larger packages. Since the hierarchic structure is created explicitly by the software designers, it can be assumed to encode key insights in the software architecture. A main feature of our visualization technique is that it places a special emphasis on these hierarchic relations and encodes them differently from non-hierarchic relations. A node-link diagram in micro/macro style that stresses the hierarchical relations may serve as a large overview map of a software project, but it still maintains details when focusing on a particular part of the layout. Much like a cartographic map that gives an overview when viewed from a distance, with only larger geographic features (or higher-level nodes) visible, but that also presents detailed information when moving closer.

1.1 Related Work

There are various methods and systems for visualizing certain aspects of a software, for two surveys see (Diehl, 2007; Diehl and Telea, 2014). Here we focus on graph-based approaches using node-link diagrams since we want to create a spatial overview map of the hierarchical structure of a software project. We note that in the literature, matrix-based and hybrid approaches are also successfully applied to software visualization, e.g., (van Ham, 2003; Rufiange et al., 2012; Abuthawabeh et al., 2013; Rufiange and Melançon, 2014).

DOI: 10.5220/0005785901530160
In *Proceedings of the 11th Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2016) - Volume 2: IVAPP*, pages 155-162
ISBN: 978-989-758-175-5

There are several previous papers that employ force-based graph layout algorithms to visualize graphs derived from software projects (Collberg et al., 2003; Beyer, 2005; Palepu and Jones, 2013). Some approaches apply clustering methods or use hierarchical information to indicate grouping patterns either by color or by spatial grouping. However, these approaches show only straight-line edges and do not properly use a containment metaphor for visualizing hierarchies. Other works focus on the hierarchical structure of software using space-filling, treemap-like 3D visualizations in the style of a city map (Wettel and Lanza, 2007), which may be enriched by showing hierarchically bundled edges on top of the city (Caserta et al., 2011). Polyptychon (Daniel et al., 2014) is an interactive tool for creating node-link diagrams of software compound graphs that focuses on dependency edges with respect to a selected view node and a local context. Hierarchical edge bundling itself has also been applied to software visualization as an independent edge layout method on top of existing hierarchical layouts (Holten, 2006) (e.g., radial, treemap, and balloon layouts).

The visualization of graphs with a hierarchy has also been investigated from a more generic graph drawing perspective, for a recent survey see (Vehlow et al., 2015). From a theoretical point of view, the most closely related problem is that of drawing clustered graphs. The question whether a planar drawing can be achieved where each cluster is a simply-connected region whose boundary is crossed at most once by every edge, is the famous c-planarity problem. This problem has been studied for more than 20 years, see (Patrignani, 2013) for an overview, but its complexity remains open and it is still an active research topic.

In the context of more applied generic algorithms, Frishman and Tal (Frishman and Tal, 2004) presented a force-based algorithm for drawing straight-line layouts of graphs with a flat clustering that forms a two-level hierarchy. Baur and Brandes (Baur and Brandes, 2008) proposed (multicircular) micro/macro layouts for visualizing hierarchic graphs. These layouts feature circular macro vertices, each representing a cluster of micro vertices and containing a circular layout of its micro vertices and edges. Unlike Frishman and Tal, their visualization makes use of edge bundling by routing micro edges through channels defined by macro edges, which reduces the clutter from connections between different clusters of the drawing. However, their approach, too, is limited to only two layers in the hierarchy. There are also approaches for visualizing graphs with more than two levels in the hierarchy, using treemap-based layouts (Muelder and Ma, 2008), using force-based layouts (Bourqui et al., 2007; Dogrusoz et al., 2009) or combinations of the two (Didimo and Montecchiani, 2012). In contrast to our proposed method they use straight-line drawings and do not support edge bundling or other edge routing techniques.

1.2 Contribution and Outline

We propose a new technique for software visualization that is designed to emphasize the inherent hierarchy of software besides showing all other edges in a bundled fashion that reduces edge clutter and avoids edge-node overlaps. It extends the idea of micro/macro layouts (Baur and Brandes, 2008) to multiple levels, uses hierarchical force-based node placement, and routes the individual (non-hierarchical) edges as polylines bundled within the wider macro edges defined at higher levels of the hierarchy.

Note that our main goal is to provide a map of a software project that displays all parts of the Software simultaneously and contains all details. Naturally, when the whole map is shown details may become very small, just like a cartographic map one has to stand close or scale the visualization and select a suitable viewport to view detail information.

We first describe our graph model and layout style in Section 2. Afterwards, we present our layout algorithm in Section 3. Finally, we present and discuss an illustrative real-world example in Section 4.

2 MODEL

In the most general setting we can model software structures as *compound graphs* (Sugiyama and Misue, 1991). A compound graph is a triple $D = (V, E, I)$, where V is a set of vertices, E is a set of (directed) *adjacency edges* on V forming the *adjacency graph* $D_a = (V, E)$, and I is another set of directed *inclusion edges* on V forming the *inclusion graph* $D_c = (V, I)$; see Fig. 1a for an example. In our model, the set V contains all relevant structural entities as vertices, e.g., packages, classes, interfaces, methods, fields, etc. The adjacency edges represent references between those entities such as method calls, field accesses, inheritance and others, whereas the inclusion edges represent containment relations of these entities as given by the software architecture, e.g., methods contained in classes contained in packages etc. Frequently, as in our case of Java source code, the inclusion graph D_c can be further restricted to being a rooted tree that naturally represents a hierarchy on V.

Generally, adjacency edges may be defined between arbitrary vertices with respect to D_c. However, since we will represent inclusion edges by geometric

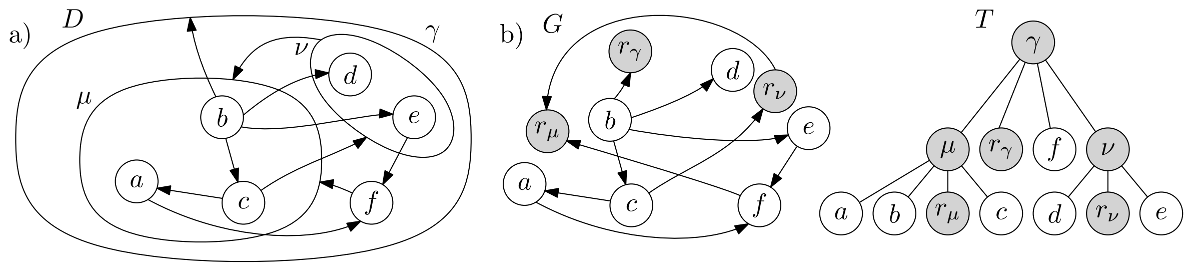


Figure 1: Example of a compound graph D (a) and the corresponding clustered graph $C = (G,T)$ (b). The inclusion edges in (a) are represented by inclusion of the vertices. The representative vertices in G as well as the cluster nodes in T are shaded. Note that edges of G only connect leaves of T.

containment, it would be difficult to draw adjacency edges between descendants and ancestors. Hence we attach in D_c to each non-leaf vertex v of D_c an additional child r_v, called *representative vertex* of v, and reconnect all edges in D_a originally incident to v to r_v instead. What we obtain now is a *clustered graph* $C = (G,T)$ consisting of the modified adjacency graph D_a as the graph G and the inclusion tree $T = D_c$, whose leaves are now exactly the vertices of G; see Fig. 1 for an example, where Fig. 1b shows the clustered graph corresponding to the compound graph from Fig. 1a. Throughout the rest of this paper we use this clustered graph C as the representation of the software structure; we refer to C as the *hierarchic software graph*.

The inclusion tree $T = (N,I)$ consists of a set of *nodes* $N = \{\mu_1,\dots,\mu_k\} \cup V$ and the set of inclusion edges I, where each μ_i is an internal node and each $v \in V$ is a leaf. We further define for each node $\mu \in N$ the set $L_\mu \subset V$ of leaves in the subtree of T rooted at μ. Let μ be an internal node of T. Then we define the *local graph* $G_\mu = (V_\mu, E_\mu)$ of μ to be the graph with vertex set $V_\mu = \{\nu \in N \mid (\nu,\mu) \in I\}$ consisting of all children of μ in T and edge set $E_\mu = \{(\nu,\eta) \mid \exists u \in L_\nu, v \in L_\eta, (u,v) \in E\}$ consisting of all edges induced by the descendants of V_μ in the adjacency graph.

Our drawing convention for visualizing hierarchic software graphs is as follows. We represent each node of T (including the leaves) as a disk. Edges of G are represented as (polygonal) curves connecting the disks of their endpoints. For simplicity, we identify vertices and edges with their corresponding disks and curves. Edges of T are represented by containment, i.e., for every internal node μ, we require that all its children are positioned inside the disk representing μ. We further require that no two disks overlap properly or, in other words, that any two disks are either disjoint or one is contained in the other. For the adjacency edges we demand that they do not pass through node disks (of course they may enter them if their destination is positioned inside). Our optimization criteria are that related vertices should be positioned close to each other and that edge crossings should be avoided. To facilitate the latter, we bundle edges with similar source or destination into channels that are routed together similar to micro/macro layouts (Baur and Brandes, 2008).

3 ALGORITHM

Our algorithm follows a two-phase approach. In the first phase, *vertex and node placement*, we perform a bottom-up traversal of the hierarchy and determine for each vertex of the graph and for each node of the hierarchy a corresponding disk. It ensures that disks are properly nested according to the inclusion edges.

In the second phase, *edge routing*, we draw the edges. We first decompose every edge into *segments*, each connecting either a child and a parent in the hierarchy or two siblings. Each segment is associated with a weight describing how many edges contain it. We then draw the segments as thick corridors (or channels) using a geometric heuristic to route around obstacles, e.g., non-incident vertices. Crossing segments that share a common endpoint are merged into a larger segment to avoid crossings. Afterwards, we draw the actual edges inside the corresponding channels.

3.1 Vertex and Node Placement

For the vertex and node placement we perform a bottom-up traversal of the cluster tree T. For each cluster μ, we assume that its children ν are already represented by disks D_ν. We assume that the leaves are represented by disks of a fixed size. When processing an inner node μ, our goal is to arrange the disks D_ν representing the children of μ in such a way that (i) they do not overlap, (ii) children that are adjacent in G_μ are close to each other, and (iii) the size of the smallest enclosing disk of the arrangement centered at the representative vertex r_μ is as small as possible (over all arrangements of the disks D_ν).

To meet these requirements, for any two vertices u and v whose corresponding disks have radius r_u and

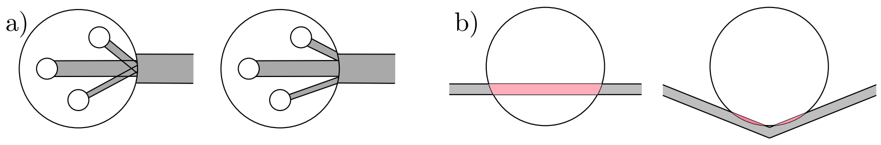


Figure 2: Illustration of channel routing. a) Offsetting channels at common ports. b) Reducing overlaps by adding bend points; iterating results in smooth routing around nodes.

r_v, we define an offset function

$$\mathrm{off}(u,v) = c_{\min} \min\{r_u, r_v\} + c_{\max} \max\{r_u, r_v\}, \quad (1)$$

where $c_{\min}, c_{\max} \geq 0$ with $c_{\min} + c_{\max} = 1$ control the weighting of the smaller and the larger disk radius. To avoid overlaps, we require that the center points of D_u and D_v have distance at least

$$d_{\min}(u,v) = r_u + r_v + b_{\min} \cdot \mathrm{off}(u,v), \quad (2)$$

where $b_{\min} \geq 1$ is a weighting parameter controlling the amount of white space between the nodes. To ensure that adjacent disks are close together, we define for adjacent nodes u and v a preferred distance

$$d_{\mathrm{pref}}(u,v) = r_u + r_v + b_{\mathrm{pref}} \cdot \mathrm{off}(u,v), \quad (3)$$

where $b_{\mathrm{pref}} \geq b_{\min}$ controls the preferred distance of adjacent vertices. The values $c_{\min}$ and $c_{\max}$ allow a better control of white space in case the two disks have very different radii.

Let Γ_μ denote the corresponding arrangement of G_μ. We then fix the radius of the disk D_μ representing μ to be slightly larger than the radius of the smallest enclosing disk of Γ_μ centered at a central node in the graph, i.e., a node with small distance to all other nodes. Once the whole tree T has been processed, the final arrangement of disks is obtained from the root drawing Γ_r (r is the root of T) by iteratively replacing the interior of disks D_μ representing non-leaf nodes μ by the corresponding arrangement Γ_μ.

For computing the layouts of the G_μ, we first compute an initial arrangement by taking a maximum weight spanning forest of G_μ (edges are weighted by the corresponding number of edges of G) and using a radial tree layout. This avoids node overlaps in the initial drawing. Afterwards, we refine the vertex positioning using the force-directed algorithm of Fruchterman and Reingold (Fruchterman and Reingold, 1991) with some simple modifications to the forces so that disk-shaped vertices of non-zero size are handled correctly. In particular, we use strong repulsive forces to ensure the minimum distance $d_{\min}(u,v)$ for all pairs $\{u,v\}$ of nodes, and attraction forces for adjacent vertices $\{u,v\}$ whose distance is larger than $d_{\mathrm{pref}}(u,v)$. For practical purposes we found that $c_{\min} = 0.05$ and $c_{\max} = 0.95$ with $2 \leq b_{\min} \leq 6$ and $b_{\mathrm{pref}} = 2b_{\min}$ gives good results; see (Schuhmacher, 2015) for more details.

3.2 Edge Routing

Let D and D' be two disjoint disks. We define the segment $s(D,D')$ connecting D and D' as the line segment between the intersections of the boundaries of D and D' and the straight-line segment connecting their centers. We consider points as disks of radius 0.

We would like to draw the edges as polygonal curves as follows. For an edge (u,v) in G let μ denote the lowest common ancestor of u and v in T and let ν and η denote the children of μ whose subtrees contain u and v, respectively. We first draw the segment $s(D_\nu, D_\eta)$ and denote its endpoints as two *ports* p_ν and p_η lying on the respective disk boundaries. It remains to draw two polygonal curves between u and p_ν as well as between v and p_η. This can be done independently of each other. We describe the curve between u and p_ν, the other curve is constructed analogously. Let $u = \nu_1, \ldots, \nu_k = \nu$ be the unique path between u and ν in T. We abbreviate the disk D_{ν_i} as D_i. We define the port p_k to be p_ν and iteratively obtain p_{i-1} as the port of $s(p_i, D_{i-1})$ on the boundary of D_{i-1}. The sequence of ports $p_1, \ldots, p_k$ defines the desired polygonal curve. Linking the three subcurves yields the polyline for edge (u,v). In the following we refer to a straight-line segment between two consecutive ports on such a polyline as a *segment*.

When applying this procedure to all edges, some segments are encountered multiple times and implicitly give rise to edge bundles. We want to avoid overplotting by separating overlapping edges. At the same time, we want to maintain the bundling as it emphasizes macro structures of the hierarchic software graph. To that end, we draw each segment as a thick curve between its ports, whose thickness is proportional to the number of edges containing it. We call these thick curves *channels* and use them as geometric containers for drawing in a second step the individual edges with a fixed pairwise offset.

The first step of our drawing procedure performs the channel routing. Initially, each channel is simply a thick straight-line segment between its ports. For each port p the thickness (or *size*) of the thickest channel at p equals the sum of the sizes of the other channels at p. If more than two channels share a port p, we offset the port positions of the smaller channels such that they do not overlap, see Fig. 2a. It may still happen that a

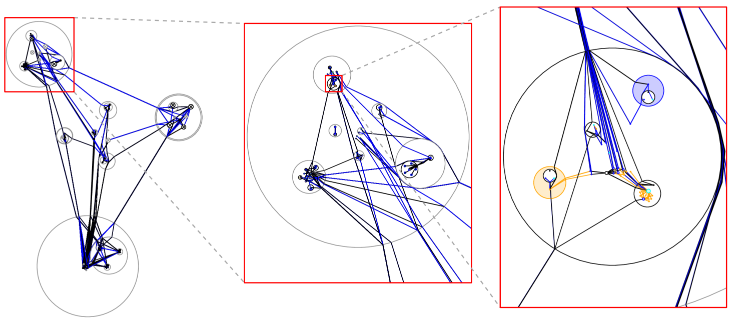


Figure 3: Visualization of a Java software project at three different zoom levels. In the top level, we see the domain model on the left, the GUI package at the bottom and a utility package in the top right. The middle zoom level shows the internal structure of the utility package. The lowest level shows the test class of the utility package.

channel crosses non-incident nodes. In order to resolve such crossings, we iteratively reroute those channels by introducing an additional bend point and moving it out of the respective node, see Fig. 2b. This is repeated until no further improvement is achieved. Finally, we apply a simple fix to resolve crossings between two neighboring channels with the same target by merging them at their first intersection into a thicker channel.

The second step routes individual edges within their induced sequence of channels. In order to avoid unnecessary crossings within the channels, we use a simple heuristic to order the outgoing edges at the vertices of G in a way that reflects as much as possible the hierarchic bundling expressed by the channels.

4 REAL-WORLD EXAMPLES

In this section we present several example visualizations of real-world software projects. Figure 3 demonstrates the use of our visualizations for visualizing a software at different levels of detail. The software has three main components: a graphical user interface, a domain model and a utility package. The top, coarsest zoom level gives an overview of the overall structure and highlights the three main packages (gray circles) and their relations. The middle view depicts the internal structure of the utility package. Here, the bundling inside the channels allows to analyze how different submodules relate to the external components. Finally, the bottom, detailed view exhibits the internal structure of a single class (black circle) composed of methods (blue disks), constructors (light-blue disks), fields (orange disks), as well as some subclasses.

Figure 4 shows a couple of classes and the corresponding test classes. The visualization clearly shows that each test classes only refers to one class.

Figure 5 shows an example, where by visual inspection of our layouts we found some code duplication in a formula parser, which lead to a refactoring.

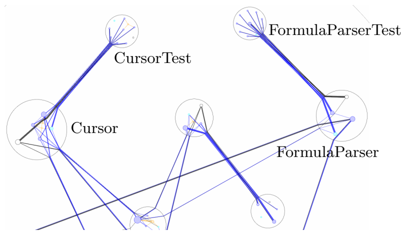


Figure 4: Visualization of classes and corresponding tests.

5 CONCLUSION

In this paper we have presented a prototype system for visualizing hierarchical software graphs as micro/macro node-link diagrams. Our focus in this paper was the presentation of a tailored algorithm and its prototypical implementation for visualizing hierarchical compound graphs as they arise in software engineering. Our current implementation serves as a proof of concept, but it can already be used to interactively explore Java software projects. Still, it must be extended in multiple ways to meet the needs of software visualization in practice. Thus the next step is to integrate it into IDEs (e.g., Eclipse) to visually support software engineering tasks, in particular to enable interactive linking between the visual representations and the corresponding source code. This includes a dynamic label placement step that shows the names of all software entities relevant at the current level of detail. Figure 7 and Figure 8 show screenshots of a preliminary integration into Eclipse for a specific software project, which also includes highlighting and a simple labeling algorithm. Further improvements on the layout quality comprise additional postprocessing steps to minimize white space and to improve edge routing by using smooth curves instead of hard bends and spreading edges more evenly at vertices.

Once this is done, a more formal validation (Seriai et al., 2014) together with software engineers and

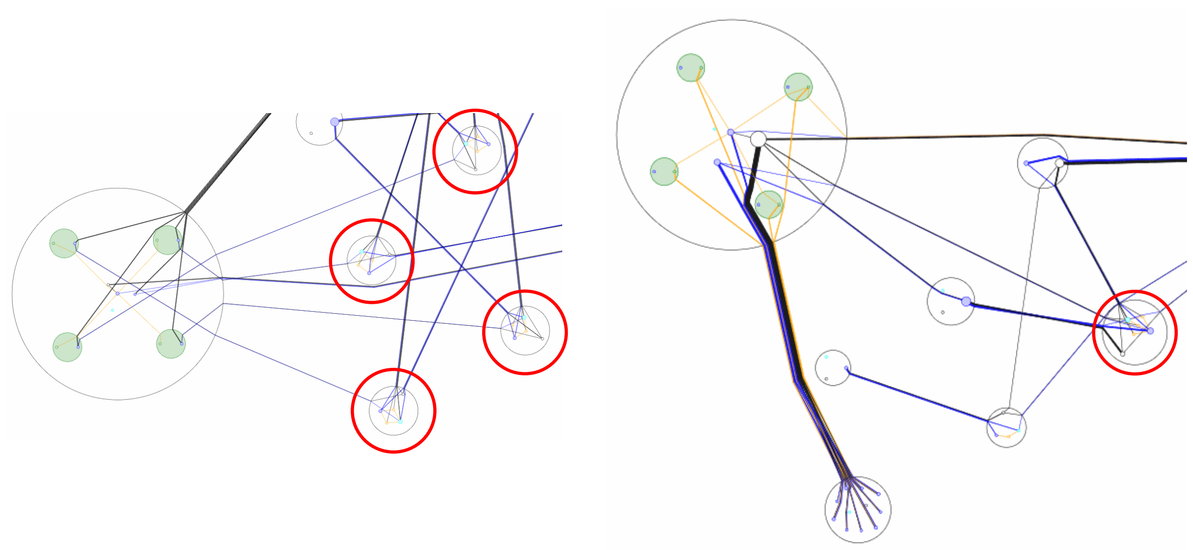


Figure 5: Visualization of a formula parser. The left side visually depicts code duplication (the four marked classes). The right side shows the same sofware project after a refactoring that removes the code duplication.

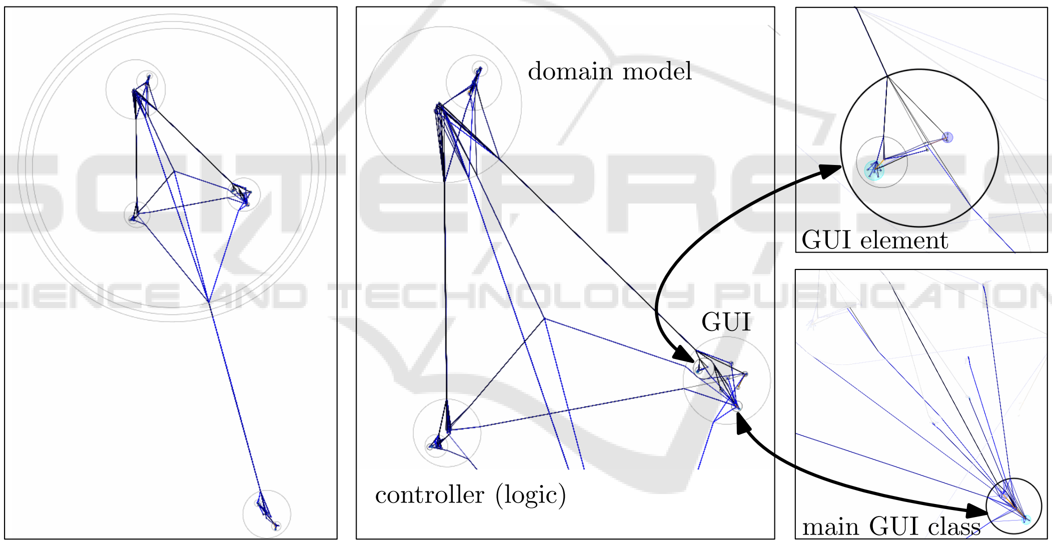


Figure 6: Visualization of a Java software project (labels have been added manually). The package in the bottom right of the overview is a utility package. The main package displays very clearly the model view controller architecture of the software. The top right shows a GUI element (actually a table view). The large node inside represents a subclass used for connecting to the model. The lower right shows a focus view of the main GUI class, where only the connections of that class are highlighted. This makes it easy to see that this class references all classes in the GUI package but does not refer to objects outside the GUI package.

comparison against existing software visualization approaches is another important step for future work.

REFERENCES

Abuthawabeh, A., Beck, F., Zeckzer, D., and Diehl, S. (2013). Finding structures in multi-type code couplings with node-link and matrix visualizations. In *Software Visualization (VISSOFT'13)*, pages 1–10.

Baur, M. and Brandes, U. (2008). Multi-circular layout of micro/macro graphs. In Hong, S.-H., Nishizeki, T., and Quan, W., editors, *Graph Drawing*, volume 4875 of *Lecture Notes in Computer Science*, pages 255–267. Springer Berlin Heidelberg.

Beyer, D. (2005). Co-change visualization. In *Proceedings of the 21st IEEE International Conference on Software Maintenance - Industrial and Tool volume, ICSM 2005, 25-30 September 2005, Budapest, Hungary*, pages 89–92.

Bourqui, R., Auber, D., and Mary, P. (2007). How to draw clustered weighted graphs using a multilevel force-directed graph drawing algorithm. In *Proc. 11th Int'l Conference on Information Visualization (IV'07)*, pages 757–764. IEEE.

Caserta, P., Zendra, O., and Bodénès, D. (2011). 3d hierarchical edge bundles to visualize relations in a software city metaphor. In *Visualizing Software for Understanding and Analysis (VISSOFT'11)*, pages 1–8. IEEE.

Collberg, C. S., Kobourov, S. G., Nagra, J., Pitts, J., and Wampler, K. (2003). A system for graph-based visualization of the evolution of software. In *Proceedings ACM 2003 Symposium on Software Visualization, San Diego, California, USA, June 11-13, 2003*, pages 77–86, 212–213.

Daniel, D. T., Wuchner, E., Sokolov, K., Stal, M., and Liggesmeyer, P. (2014). Polyptychon: A hierarchically-constrained classified dependencies visualization. In *Software Visualization (VISSOFT'14)*, pages 83–86.

Didimo, W. and Montecchiani, F. (2012). Fast layout computation of hierarchically clustered networks: Algorithmic advances and experimental analysis. In *Proc. 16th Int'l Conf. Information Visualization (IV'12)*, pages 18–23. IEEE.

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

Diehl, S. and Telea, A. C. (2014). Multivariate networks in software engineering. In Kerren, A., Purchase, H. C., and Ward, M. O., editors, *Multivariate Network Visualization*, volume 8380 of *LNCS*, chapter 2, pages 13–36. Springer International Publishing.

Dogrusoz, U., Giral, E., Cetintas, A., Civril, A., and Demir, E. (2009). A layout algorithm for undirected compound graphs. *Information Sciences*, 179(7):980–994.

Frishman, Y. and Tal, A. (2004). Dynamic drawing of clustered graphs. In *Information Visualization (INFOVIS'04)*, pages 191–198. IEEE.

Fruchterman, T. M. J. and Reingold, E. M. (1991). Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164.

Holten, D. (2006). Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):741–748.

Muelder, C. and Ma, K.-L. (2008). A treemap based method for rapid layout of large graphs. In *Proc. IEEE Pacific Visualization Symposium (PacificVis'08)*, pages 231–238.

Palepu, V. K. and Jones, J. A. (2013). Visualizing constituent behaviors within executions. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT), Eindhoven, The Netherlands, September 27-28, 2013*, pages 1–4.

Patrignani, M. (2013). *Handbook of Graph Drawing and Visualization*, chapter Planarity Testing and Embedding, pages 1–42. Discrete Mathematics and its Applications. CRC Press.

Rufiange, S., McGuffin, M. J., and Fuhrman, C. P. (2012). Treematrix: A hybrid visualization of compound graphs. *Computer Graphics Forum*, 31(1):89–101.

Rufiange, S. and Melançon, G. (2014). Animatrix: A matrix-based visualization of software evolution. In *Software Visualization (VISSOFT'14)*, pages 137–146.

Schuhmacher, A. (2015). Software visualization via hierarchic graphs. Master's thesis, Karlsruhe Institute of Technology (KIT).

Seriai, A., Benomar, O., Cerat, B., and Sahraoui, H. (2014). Validation of software visualization tools: A systematic mapping study. In *Software Visualization (VISSOFT'14)*, pages 60–69.

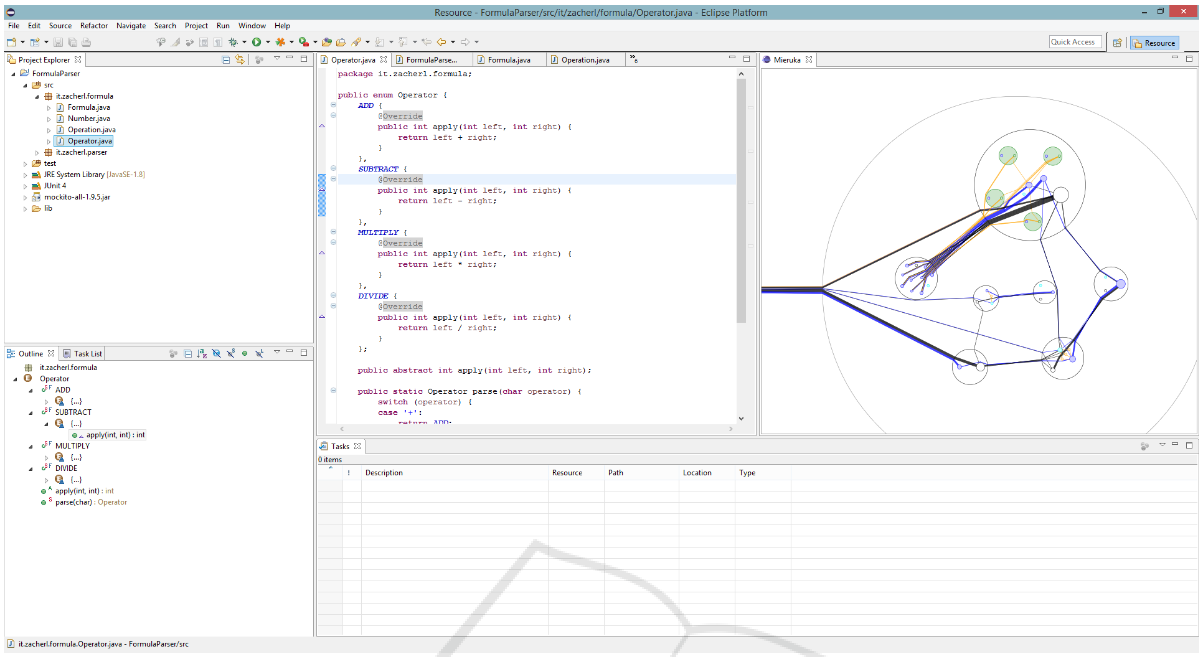
Sugiyama, K. and Misue, K. (1991). Visualization of structural information: Automatic drawing of compound digraphs. *IEEE Trans. Syst. Man Cybern.*, 21(4):876–892.

van Ham, F. (2003). Using multilevel call matrices in large software projects. In *Information Visualization (INFOVIS'03)*, pages 227–232.

Vehlow, C., Beck, F., and Weiskopf, D. (2015). The state of the art in visualizing group structures in graphs. In Borgo, R., Ganovelli, F., and Viola, I., editors, *Eurographics Conference on Visualization (EuroVis'15)*, STARs.

Wettel, R. and Lanza, M. (2007). Visualizing software systems as cities. In *Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT 2007, 25-26 June 2007, Banff, Alberta, Canada*, pages 92–99.

APPENDIX


Resource - FormulaParser/src/it/zacherl/formula/Operator.java - Eclipse Platform

File Edit Source Refactor Navigate Search Project Run Window Help

Project Explorer
```
FormulaParser
  src
    it.zacherl.formula
      Formula.java
      Number.java
      Operation.java
      Operator.java
    it.zacherl.parser
  test
  JRE System Library [JavaSE-1.8]
  JUnit 4
  mockito-all-1.9.5.jar
  lib
```

Outline | Task List
```
it.zacherl.formula
Operator
  ADD
    {...}
  SUBTRACT
    {...}
      apply(int, int) : int
  MULTIPLY
    {...}
  DIVIDE
    {...}
  apply(int, int) : int
  parse(char) : Operator
```

Operator.java | FormulaParse... | Formula.java | Operation.java
```java
package it.zacherl.formula;

public enum Operator {
    ADD {
        @Override
        public int apply(int left, int right) {
            return left + right;
        }
    },
    SUBTRACT {
        @Override
        public int apply(int left, int right) {
            return left - right;
        }
    },
    MULTIPLY {
        @Override
        public int apply(int left, int right) {
            return left * right;
        }
    },
    DIVIDE {
        @Override
        public int apply(int left, int right) {
            return left / right;
        }
    };

    public abstract int apply(int left, int right);

    public static Operator parse(char operator) {
        switch (operator) {
        case '+':
```

Mieruka

Tasks
0 items

!	Description	Resource	Path	Location	Type

it.zacherl.formula.Operator.java - FormulaParser/src

Figure 7: Integration of our visualization into Eclipse for a specific software project.

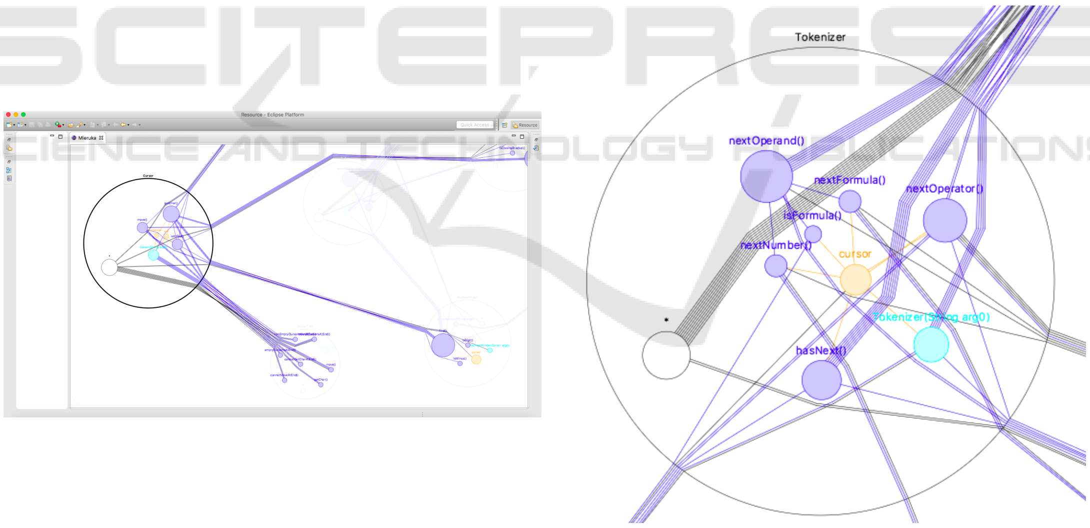


Figure 8: The left shows a package in the eclipse integration and highlights only the objects that are referenced from within this package. The right demonstrates the inclusion of a simple algorithm that labels the entities shown in the visualization.