A Survey on Interactive Grouping and Filtering in Graph-based Software Visualizations

Ivica Aracic Thorsten Schäfer Mira Mezini Klaus Ostermann

Software Modularity Lab
Department of Computer Science
Darmstadt University of Technology
aracic@stribor.de, {schaefer, mezini, ostermann}@informatik.tu-darmstadt.de

Abstract

Interactive grouping and filtering in software visualization tools are essential mechanisms enabling the users to build views that match the information needs of their software comprehension task at hand. In this paper we systematically survey these mechanisms in eight graph-based software visualization tools.

1 Introduction

Graph-based Software visualization can help in reasoning about software systems, which is an important prerequisite for almost any software development task. An important goal of software visualization is to provide, what we call WYSIWYN (What You See Is What You Need) views that

1. match the structure of the problem at hand, i.e., present the information at the right level of abstraction, and
2. contain only relevant information

Such views are desirable as they ease the interpretation of the view by (1) reducing the gap between the structure of the problem at hand and the structure of the view [3], and (2) reducing the amount of presented information, thus, avoiding information overload [10, 15].

Hence, interactive grouping and filtering operations on graph-based views are required. Grouping addresses the first property of WYSIWYN views by enabling the user to group elements of the view according to a common property, thus introducing new abstractions to the view and closing the gap between the view and the problem structure. Filtering supports the second property by allowing the user to remove irrelevant information from the view. Finally, the interactivity ensures that the user is able to incrementally refine an initial view to match the needs of the task at hand. This is an important property for general purpose tools. Because of the large class of potential software comprehension tasks that they aim to cover, no automatism can be provided that always constructs a perfect WYSIWYN view. In most cases a user-driven refinement of the initial view will be required.

The main contribution of this paper is in providing a minimal set of grouping and filtering operations on hierarchically structured (nested) graph models and in surveying eight state-of-the-art tools with respect to their support for these operations.

The rest of the paper is structured as follows. In Sec. 2, we formalize the nested graph model and derive from it four primitive operations and their corresponding inverses. Sec. 3 demonstrates the application of the derived operations by means of an example. Sec. 4 presents the survey of eight software visualization tools with respect to their interactive support for the stated primitive operations. Finally, we conclude with Sec. 5.

2 Filtering and Grouping in Nested Graphs

A widely used metaphor in software visualization is the nested graph. A nested graph is a hierarchically structured graph with a hierarchy of nodes and a hierarchy of edges. Such graphs are suitable for visualizing complex, hierarchically structured data like imposed by object-oriented software systems.

In this section, we formalize the nested graph and derive four primitive operations from it that support grouping and filtering of graph's elements, nodes and edges respectively. The operations are primitive in the sense that they can not be reconstructed by an arbitrary sequence of other primitive operations.

Our formalism differs from other nested graph formalisms (e.g. [8]) in two points: (1) to best of our knowledge no formalism, that we are aware of, considers the node and the edge hierarchy as two symmetric structures and (2) no formalism considers the grouping and filtering operations on nested graphs.

Nested Graph Model

Let N be a finite set of nodes and $E \subseteq N \times N$ a set of edges. A *node nesting* c_N is a finite mapping $c_N : N \to \mathcal{P}(N)$ and an edge nesting c_E is a finite mapping $c_E : E \to \mathcal{P}(E)$. For a node nesting or an edge nesting ($X = N$ or $X = E$), the set of (direct and indirect) children of a node or edge x is the smallest set $c_X^+(x)$ satisfying

$$c_X^+(x) = c_X(x) \cup \bigcup_{x' \in c_X(x)} c_X^+(x')$$

We define a reflective variant of this set as:

$$c_X^*(x) = \{x\} \cup c_X^+(x).$$

Finally, we define a *nested graph* as a tuple

$G = (N, E, c_N, c_E, n_{root})$ such that

1. Nodes and edges are organized in a strict hierarchy: For all $n \in N \setminus \{n_{root}\}$ there is a unique $n' \in N$ such that $n \in c_N(n')$. Analogously, for all $e \in E \setminus \{(n_{root}, n_{root})\}$ there is a unique $e' \in E$ such that $e \in c_E(e')$. The designated node n_{root} is the root node: $c_N^*(n_{root}) = N$. Furthermore, $(n_{root}, n_{root}) \in E$ and $c_E^*(n_{root}, n_{root}) = E$.

2. The edge hierarchy must conform to the node hierarchy: $\forall e = (n_1, n_2), e' = (n_1', n_2') \in E . e' \in c_E^+(e) \Rightarrow n_1' \in c_N^*(n_1), n_2' \in c_N^*(n_2)$.

Note that invariant in (2) is required in order to make composite edges interpretable in the context of a node hierarchy. While the criteria how nodes are grouped is unrestricted, the grouping criteria of edges must ensure that all contained edges in a composite have the common property defined in (2). I.e., for all children of an edge, the source node of the child edge is the source node of the containing edge or a child of it, and the target node of the child edge is the target node of the containing edge or a child of it.

Group/Ungroup

The edge and node hierarchy of the nested graph corresponds to the encoding of the problem structure. To be able to match the structure of the task at hand, as required for WYSIWYN views, the user needs means for tailoring these two hierarchies. Hence, the two hierarchies form the first two dimensions along which the nested graph can be tailored. Two primitive operations and their inverses are derived: $group_N / ungroup_N$ for nodes and $group_E / ungroup_E$ for edges respectively.

For both hierarchies, we formalize these in the following. The preconditions of the operations assure that the defined nested graph invariants are preserved.

Operation: $group_N : G \times N \times \mathcal{P}(N) \to G$
Effect: A new node n is added and replaces p as common parent of all nodes in ns.
Precondition: $ns \subseteq c_N(p)$ and $n \notin N$.

$$group_N((N, E, c_N, c_E, n_{root}), n, ns) = (N \cup \{n\}, E, c_N', c_E, n_{root})$$

$$\text{where} \quad c_N'(x) = \begin{cases} (c_N(x) \setminus ns) \cup \{n\} & \text{if } x = p \\ ns & \text{if } x = n \\ c_N(x) & \text{else} \end{cases}$$

Operation: $ungroup_N : G \times N \to G$
Effect: The node is removed and all its children are added to its parent.
Precondition: No edge in E touches n and $n \neq n_{root}$.

$$ungroup_N((N, E, c_N, c_E, n_{root}), n) = (N \setminus \{n\}, E, c_N', c_E, n_{root})$$

$$\text{where} \quad c_N'(x) = \begin{cases} c_N(x) \cup c_N(n) & \text{if } n \in c_N(x), \\ c_N(x) & \text{else} \end{cases}$$

Operation: $group_E : G \times N \times N \times \mathcal{P}(E) \to G$
Effect: A new edge between n_1 and n_2 is added and replaces p as common parent of all edges in es.
Precondition: $(n_1, n_2) \notin E, es \subseteq c_E(p), \forall (n, n') \in es : n \in c_N^*(n_1) \wedge n' \in c_N^*(n_2)$

$$group_E((N, E, c_N, c_E, n_{root}), n_1, n_2, es) = (N, E \cup \{(n_1, n_2)\}, c_N, c_E', n_{root})$$

$$\text{where} \quad c_E'(x) = \begin{cases} (c_E(x) \setminus es) \\ \quad \cup \{(n_1, n_2)\} & \text{if } x = p \\ es & \text{if } x = (n_1, n_2) \\ c_E(x) & \text{else} \end{cases}$$

Operation: $ungroup_E : G \times E \to G$
Effect: Removes an edge and makes all child edges top-level edges.
Precondition: $e \neq (n_{root}, n_{root})$

$$ungroup_E((N, E, c_N, c_E, n_{root}), e) = (N, E \setminus \{e\}, c_N, c_E', n_{root})$$

$$c_E'(x) = \begin{cases} c_E(x) \cup c_E(e) & \text{if } e \in c_E(x), \\ c_E(x) & \text{else} \end{cases}$$

Filter/Unfilter

The second property of WYSIWYN views requires the ability to filter irrelevant parts of the graph. Hence, for each hierarchy we define one dimension along which the filtering of nodes or edges can be applied. We define primitive operations $filter_N/unfilter_N$ for nodes and $filter_E/unfilter_E$ for edges, respectively.

To be able to filter a nested graph, we define a filtered nested graph $G_{filtered}$ as a nested graph $G = (N, E, c_N, c_E, n_{root})$ together with two filter sets $f_N \subseteq N$ and $f_E \subseteq E$. The grouping and ungrouping operations remain unchanged, except that an ungroup operation removes the corresponding node or edge from f_N or f_E, if necessary. For filtered nested graphs, we define four primitive operations, $filter_N(G_{filtered}, n)$, $unfilter_N(G_{filtered}, n)$, $filter_E(G_{filtered}, e)$, $unfilter_E(G_{filtered}, e)$, which just add or remove the respective node or edge from f_N or f_E.

For a filtered nested graph, we define a node n to be *visible*, if n is n_{root}, or $n \notin f_N$ and $n \in c_N(n')$ of a visible node n'. An edge $e = (n_1, n_2)$ is visible, if e is (n_{root}, n_{root}), or n_1 and n_2 are visible, $e \notin f_E$, and $e \in c_E(e')$ of a visible edge e'.

3 Example

In order to exemplify the usage of the primitive operations defined, we present how the ispace[1] [2] tool was used in a software comprehension task which arised during a quality assessment of the prefuse project [13]. In concrete, the prefuse's dependencies were investigated with respect to the package design principles stated by Martin [9]. Prefuse [13] is a Java-based toolkit for building interactive information visualization applications. It consists of approximately 17k non-commented, non-blank lines of code in 191 classes. For our experiment, we used version v20050401-alpha.

Prefuse is a framework for visualizing data models as graphs. For each model element, a corresponding visual item exists that contains information about the graphical representation. The visual items form a type hierarchy containing the classes `VisualItem`, `NodeItem`, `EdgeItem`, and `AggregateItem`. The classes are used by many other classes within prefuse. To ensure maintainability, we want the classes that visual items depend on to be stable; otherwise, changes there could propagate via the visual items to large parts of prefuse.

For this purpose, we needed to analyze the dependencies of the visual item classes. We needed to identify on which classes they depend and in which packages these classes are contained.

[1] http://ispace.stribor.de/

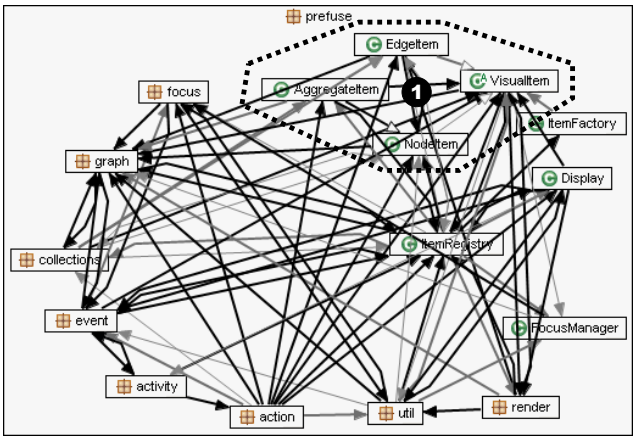

Figure 1. Initial view

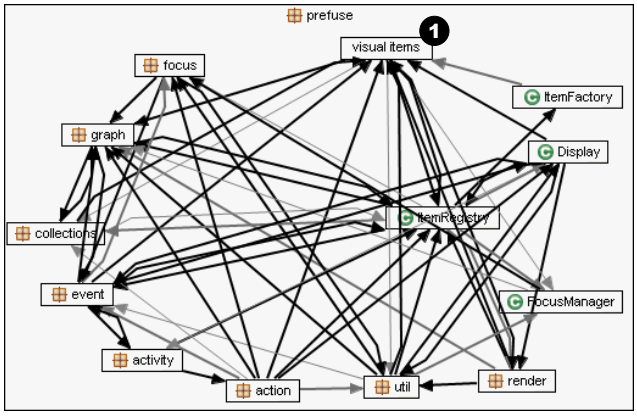

Figure 2. Grouping the visual items

In the following we describe the steps performed with the tool. Only primitive operations proposed in Sec. 2 are used. Figures 1 to 8 show the successive steps performed with ispace in order to come from the initial view to the final WYSIWYN view. The initial view (Fig. 1) contains the 16 top-level elements (8 packages and 8 classes) of prefuse with all dependencies between them. Note that even a view with only 16 elements easily gets cluttered.

Step 1 – $group_N$**:** In the first step, the four classes comprising the visual item hierarchy (index 1 in Fig. 1) were grouped together yielding the composite indexed with 1 in Fig. 2.

Step 2 – $filter_N$**:** Next, all packages and classes that are not used by the visual item hierarchy are filtered. In ispace, this is achieved by selecting corresponding elements and apply the filter operation. The resulting view is depicted in Fig. 3.

Step 3 – $filter_E$**:** While the removal of irrelevant nodes made the view more clear, in the resulting view there are still many relations which are not relevant for the task at

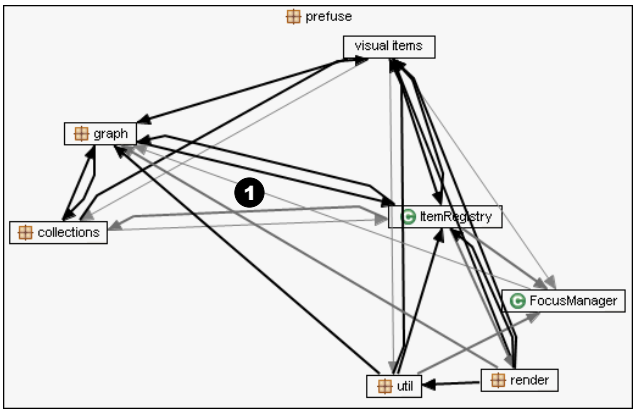

Figure 3. Filtering irrelevant elements

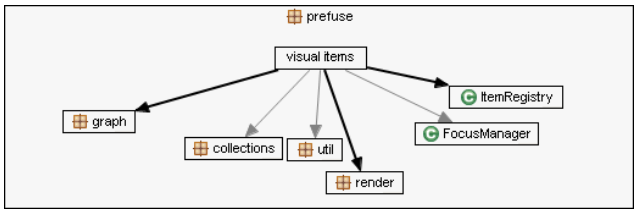

Figure 4. Filtering irrelevant relations

hand. As non-relevant, we consider all relations not starting at "visual items", e.g., the relations around index 1 in Fig. 3. Hence, in this step all relations not starting at "visual items" were selected and filtered, leading to the view depicted in Fig. 4).

Step 4 – $filter_N$**:** To investigate which classes within a package are used by visual items, the children of the used packages were unfiltered. Fig. 5 shows the view in which this transformation is applied to the package `render` (index 1 in Fig. 5).

Step 5 – $group_E$**:** In a follow up step, the hierarchy of the edges starting at `visual items` was tailored by ungrouping the composite relations ending at any package. Fig. 6 (cf. indexes 1 and 2) shows the result of ungrouping the composite edge connecting the composite node for "visual items" and the package `render` (index 2 in Fig. 5).

Step 6 – $filter_N$**:** Now the dependencies of visual item were explicit and all classes not used were filtered. Fig. 7 shows the result exemplary for the `render` package (index 1).

After performing steps four to six on the other packages, the final WYSIWYN view is created (Fig. 8). It exactly matches the structure of the problem by containing representations for: the visual item type hierarchy (index 1), classes used by the visual item hierarchy (e.g., index 2), use relations (e.g., index 3), and the enclosing package of the used classes (e.g., index 4). Moreover, all irrelevant information in the view is filtered away. This includes the unused nodes and all relations except the desired relations starting at the `visual items` node.

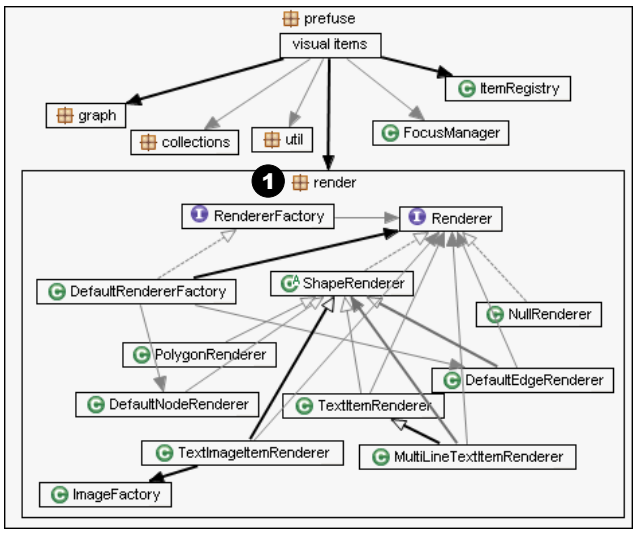

Figure 5. Unfiltering a composite's children

Note that even for this very simple task we required all proposed operations defined in Sec. 2.

4 Survey

Many software visualization tools support developers in their comprehension tasks and most of them offer some interactive support for grouping and filtering. For the eight surveyed tools, we summarize the support provided as shown in Table 1. A cross indicates that, for a given dimension, the tool directly or indirectly implements the proposed primitive operations. Indirectly means that there is a combination of available operations that yields the effect of the proposed tailoring operations. For instance, none of the tools directly supports unfiltering of a single node, n. However, the same effect can be achieved by two steps: (a) unfiltering all nodes, and (b) subsequently filtering all of them but n.

In the following, we elaborate on the surveyed tools first with respect to their support for grouping/ungrouping and filtering/unfiltering, respectively.

$group_N$

Tools that support developers in gaining high-level overviews of software tend to provide support for grouping and ungrouping of nodes and edges ($group_N$, $group_E$). For instance, Gorton and Zhu [4] present an experience report about architecture reconstruction tools. They found

Tool	$group_N/$ $ungroup_N$	$group_E/$ $ungroup_E$	$filter_N/$ $unfilter_N$	$filter_E/$ $unfilter_E$
Rigi [11]	×	–	×	–
SHriMP [18]	–	×	×	–
Relo [16]	–	–	×	×
Softwarenaut [7]	–	–	×	×
CodeCrawler [6]	×	–	–	–
Sotograph [17]	–	–	×	–
Bauhaus [1]	×	–	×	×
ispace [2]	×	×	×	×

Table 1. Interactive Tailoring Support in State-of-the-Art

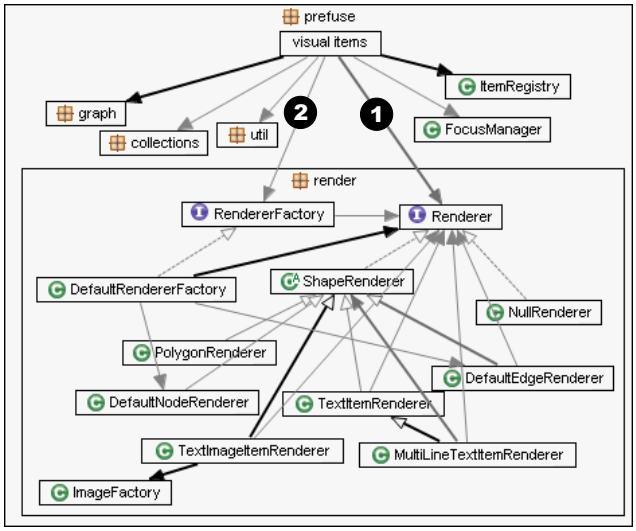

Figure 6. Ungrouping a relation

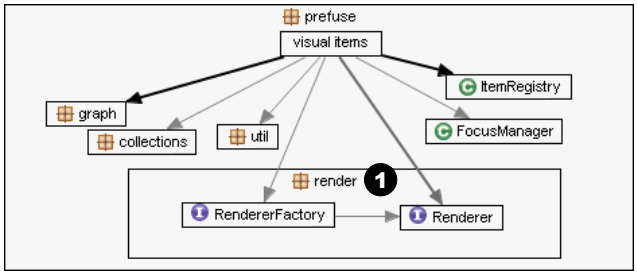

Figure 7. Filtering non-used elements

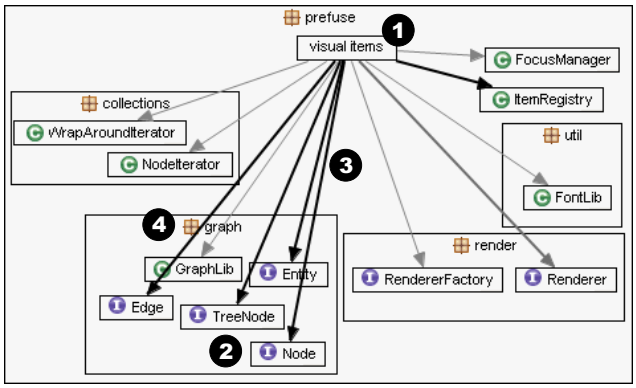

Figure 8. Final WYSIWYN view

that flexible abstractions are needed and the mapping information between different layers of abstraction needs to be maintained. The model for software visualization tools presented by Pacione [12] also contains several levels of abstraction, needed to support a broad range of software comprehension tasks.

Beside ispace, several of the surveyed tools provide support for interactively building arbitrary groups of nodes, e.g., Rigi [11], Codecrawler [6], and Bauhaus [1]. They enable the user to interactively group nodes in the presented graph and build higher level abstraction useful for solving the problem at hand.

The second group of tools including SHriMP [18], Softwarenaut [7], and Sotograph [17] enable the user to initially define an arbitrary grouping of nodes at model construction time, however, given some initial decomposition of the system, no means are provided for interactive refinement of it.

This hinders the user in interactively constructing a view that matches the structure of his problem. Considering our example from Sec. 3, the SHriMP user for instance is not able to interactively construct a node representing all visual items (Fig. 9, index 1). Hence, dependencies starting at visual items could also not be grouped. This cluttered the view with irrelevant details of the visual items hierarchy. For instance, instead of representing the dependency between the visual item hierarchy (index 3) and the `ItemRegistry` (index 5) with a single edge, several edges exist which can cause cognitive overload. Further, the relations of each element of the visual item hierarchy need to be aggregated mentally: to understand all dependencies of

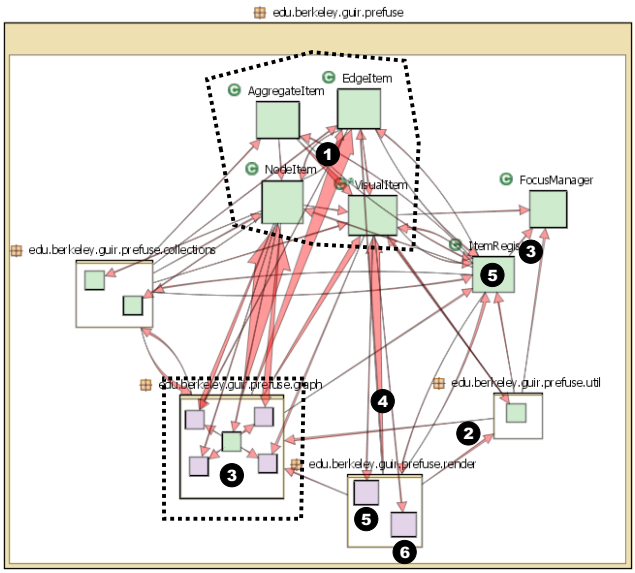

Figure 9. Task 1 (SHriMP)

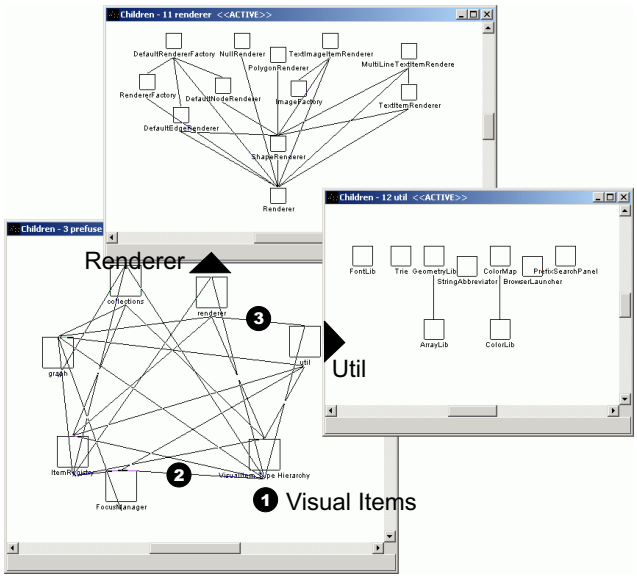

Figure 10. Task 1 (Rigi)

the visual item hierarchy, the outgoing dependencies of all visual item element have to be considered on its own.

The third group, which was only comprised of Relo [16], had no support for user defined groupings of software elements at all. The grouping of its elements is completely fixed to the primary decomposition of the visualized system (e.g.: packages, classes, methods, and fields).

$group_E$

Analogously to grouping of nodes, a flexible grouping of edges is required. Often lower-level relations need to be aggregated at higher-level of abstractions. Moreover, details of such aggregations need to be revealed as required by the task. The support for this is often neglected as we can see in the number of tools which support this operation (only SHriMP and ispace).

The following two example demonstrate the problems caused by missing support for grouping and un-grouping of edges.

For instance, Rigi provides support for grouping nodes, which enabled to construct a node representing all visual items (Fig. 10, index 1) and a composite edge representing its grouped relations (e.g., index 2)[2]. However, it lacks support for ungrouping edges. For instance, it is not possible to ungroup the composite edges starting at `visual items` in order to find out which of the contained classes were actually used by the hierarchy.

[2]Please note that directed edges in Rigi have no arrowheads. Their direction is encoded so that the source side of the the edge always starts at the bottom of the source node and the target side always ends at the top of the target node.

Another example is Relo, which lacks means for grouping lower-lever relations. As shown in Fig. 11, low-level elements like methods and method-calls need to be visualized in order to see the relation between two classes. For instance, to visualize the relation between `NodeItem` and `TreeNode`, the low-level relations between the classes' methods need to be shown (Fig. 11, index 2; note that we did not expand all relations for better readability of the diagram).

$filter_N$

Another way to cope with the complexity in software visualization tools is to remove irrelevant information from the view. Corresponding filtering mechanisms are available in many tools.

For instance, exploration tools [5, 14, 16] naturally support filtering of nodes ($filter_N$). Typically, a developer starts with a given program element and explores its neighborhood by following different kinds of relations. Hence, only some elements are visualized instead of the whole software. Tailoring along this dimension is also supported by most other visualization tools.

$filter_E$

Most tools enable filtering all relations of a specific type, e.g., all method-call or inheritance relationships; also filtering of edges when one of their adjacent nodes is removed to ensure a consistent graph is often supported. What is often missing, however, is a possibility to filter individual edges, or a set of edges selected by some arbitrary criteria.

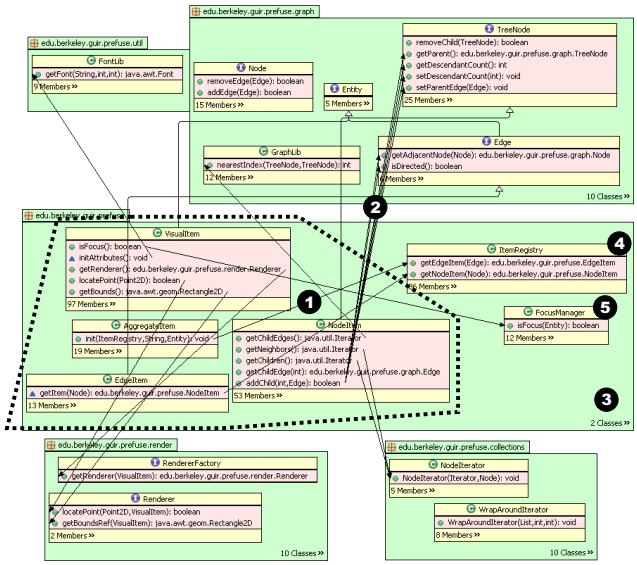

Figure 11. Task 1 (Relo)

An exception is Relo [16] which enables filtering of both individual nodes ($filter_N$) and individual edges ($filter_E$).

In Fig. 9, we can observe the information overload caused by the missing support for filtering edges in SHriMP. For example, the dependencies shown in the view around indexes 2 and 3 are completely irrelevant for the task at hand. Especially the `ItemRegistry` class (index 5) significantly contributed to the cluttering of the view, because it is related to almost any other element. Filtering this node is not a solution, because this would also remove the relevant information that this class is used by the visual item hierarchy.

Survey Summary

Most of the surveyed tools lack support for $group_N$, $group_E$, and $filter_E$. In the following we summarize the problems caused by this.

Lack of $group_N$: Without means to group elements, one cannot abstract from the individual elements. This implies a) that the view is cluttered with too many details and b) that the structure of the visualization does not match directly to the structure of the problem, which hinders comprehension.

Lack of $group_E$: The effect of lacking means to group edges is similar as for nodes: instead of a single composite edge, many low-level edges are visualized, which clutter the view. Hence, the right information is shown, but at the wrong level of abstraction.

Lack of $filter_E$: If no means exist to filter individual edges, irrelevant information cannot be removed from the view which also contributes to the clutter.

Support for $filter_N$ is provided by almost any of the surveyed tools and does not require further discussion.

5 Conclusion

Support for interactive grouping and filtering is one important requirement on software visualization tools. Grouping enables the user to close the gap between the structure of the problem and the structure of the view. Filtering helps in removing irrelevant information from the view. Interactivity enables users to incrementally refine an initial view on-the-fly as required by the current task.

In this paper we systematically formalized this requirement by defining the nested graph model and deriving four primitive operations on this model.

Furthermore, we surveyed 8 software visualization tool with respect to the proposed primitive operations. We observed that none of the investigated tools provides full support for all identified operations. Nevertheless, evidence for each operation was found at least in one of surveyed tools. By means of an example we show that such tools fail in providing what we call WYSIWYN views.

We conclude this paper with a recommendation for tool developers to close the gap identified by the survey and provide complete support for all proposed operations.

References

[1] G. V. Aoun Raza and E. Pldereder. Bauhaus - a tool suite for program analysis and reverse engineering. In *Reliable Software Technologies, Ada-Europe 2006, LNCS(4006)*, pages 71–82, 2006.

[2] I. Aracic and M. Mezini. Flexible abstraction techniques for graph-based visualizations. `http://www.sciences.univ-nantes.fr/lina/atl/www/papers/eTX2006/02-IvicaAracic.pdf`, 2006.

[3] D. J. Gilmore and T. R. Green. Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies*, 21(1):31–48, 1984.

[4] I. Gorton and L. Zhu. Tool support for just-in-time architecture reconstruction and evaluation: an experience report. In *Proceedings of the International Conference on Software Engineering*, pages 514–523. ACM Press, 2005.

[5] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *Proceedings of the International Conference on Aspect-oriented Software Development*, pages 178–187. ACM Press, 2003.

[6] M. Lanza. Codecrawler—lessons learned in building a software visualization tool. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 409–418. IEEE Computer Society, 2003.

[7] M. Lungu and M. Lanza. Softwarenaut: Exploring hierarchical system decompositions. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 351–354. IEEE Computer Society, 2006.

[8] A. J. Malton and R. C. Holt. Boxology of nba and ta: A basis for understanding software architecture. In *Proceedings of the Working Conference on Reverse Engineering*, pages 187–195. IEEE Computer Society, 2005.

[9] R. C. Martin. *Agile Software Development. Principles, Patterns, and Practices*. Prentice Hall, 2002.

[10] G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63:81–97, 1956.

[11] H. A. Müller and K. Klashinsky. Rigi – a system for programming-in-the-large. In *Proceedings of the International Conference on Software Engineering*, pages 80–86. IEEE Computer Society, 1988.

[12] M. Pacione. *A novel software visualization model to support object-oriented program comprehension*. PhD thesis, University of Strathclyde, Glasgow, 2005.

[13] Prefuse. `http://www.prefuse.org`.

[14] T. Schäfer, M. Eichberg, M. Haupt, and M. Mezini. The SEXTANT software exploration tool. *IEEE Transactions on Software Engineering*, 32(9):753–768, 2006.

[15] J. Sillito, K. De Volder, B. Fisher, and G. Murphy. Managing software change tasks: An exploratory study. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 23–32. IEEE Computer Society, 2005.

[16] V. Sinha, D. Karger, and R. Miller. Relo: Helping users manage context during interactive exploratory visualization of large codebases. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*, pages 187–194. IEEE Computer Society, 2006.

[17] Sotograph. `http://www.sotograph.com`.

[18] M.-A. D. 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.