SN Computer Science (2022) 3:511
https://doi.org/10.1007/s42979-022-01404-6

ORIGINAL RESEARCH

Static and Dynamic Dependency Visualization in a Layered Software City

Veronika Dashuber[1] · Michael Philippsen[2]

Received: 23 September 2021 / Accepted: 6 September 2022 / Published online: 10 October 2022

Abstract
A Software City is an established way to visualize metrics such as the test coverage or complexity. As current layouting algorithms are mainly based on the static code structure, dependencies that are orthogonal to this structure often clutter the visualization and are hard to grasp. This paper applies layered graph drawing to layout a Software City in 3D. The proposed layout takes both the dependencies and the static code structure into account. While having the static dependencies encoded in the layout, we can additionally display dynamic dependencies as arcs atop the city in the night view of the Layered Software City. By applying a trace clustering technique we can further reduce the number of shown arcs. We evaluate the advantages of our layout over a classic layouting algorithm in a controlled study on a real-world project and also report on a short study that evaluates the visualization of dynamic dependencies. The source code of the layouting algorithm and the raw data of the quantitative evaluations are available from https://github.com/qaware/holoware-software-city.

Keywords Software City · Layouting algorithm · Layered graph drawing · Dependency analysis · Architecture comprehension · Trace clustering

Introduction

The IT labor market is becoming more and more flexible and both projects and employees change frequently, while complex software systems with more than 200k lines of code have a long service life and cause significant efforts for understanding software in development and maintenance projects [1]. Hence, visualization tools that help developers to sooner have a correct understanding of the software and its behavior (dynamic processes during program execution) increase productivity.

This article is part of the topical collection "Computer Vision, Imaging and Computer Graphics Theory and Applications" guest edited by Jose Braz, A. Augusto Sousa, Alexis Paljic, Christophe Hurter and Giovanni Maria Farinella.

✉ Veronika Dashuber
veronika.dashuber@qaware.de

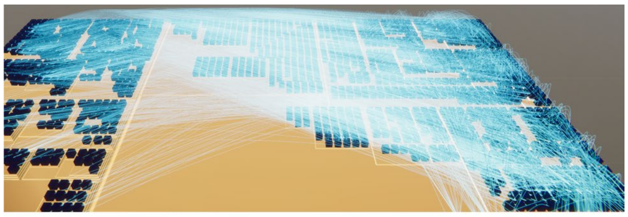
Michael Philippsen
michael.philippsen@fau.de

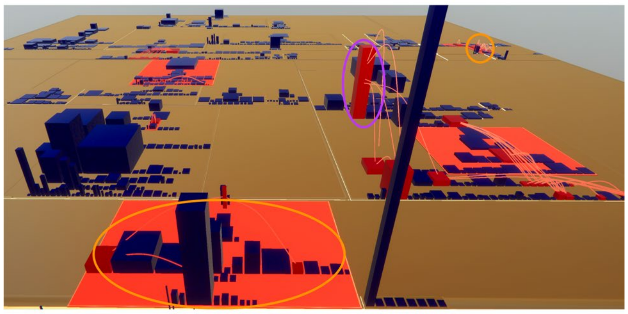
1 QAware GmbH, Aschauer Str. 32, 81549 Munich, Germany

2 Programming Systems Group, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Martensstr. 3, 91058 Erlangen, Germany

Software visualizations can cover the static structure of the source code, the behavior, or the evolution, i.e., the changes of the structure over time [2]. Regardless of which aspects are visualized, to make the abstract software artifacts easier to understand for humans, they are often mapped to familiar real-world metaphors [3]. Several controlled experiments have shown that the metaphor of a city is well suited [4–6]. It mainly focuses on the static aspects and represents components (e.g., classes) as buildings and shows containers of components (e.g., packages or modules) as city districts. There are Software Cities that also cover dynamic or evolutionary aspects. The general principle is that the hierarchical structure of the components (e.g., package → subpackage → class) is used to map artifacts to the floorplan of the city. Proximity in the source code results in proximity in the city, but not the other way round.

Nested TreeMaps and Street Views are well-known layouting techniques ("Related Work" discusses them in some detail). These layouts only consider the hierarchical code structure, i.e., contains relations. Typically, extensions visualize other dependencies among the components as arcs atop the buildings. This often leads to dependency arcs that are scattered across the entire visualization, more overwhelming than helping to understand them. To motivate our layout,


(a) TreeMap layout with all dependency arcs.


(b) Our layered layout. Most dependencies encoded in the layers; only possible architecture violation arcs remain.

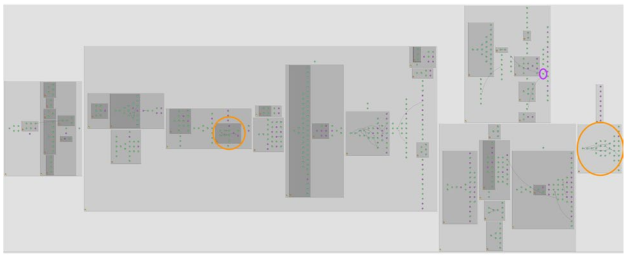

(c) Structure101 2D visualization with possible architecture violation arrows.

Fig. 1 Visualization of the Spring core source code

Fig. 1a visualizes the "core" module of the open source Spring framework[1] with the TreeMap layout, drawing all usage/invocation dependencies as arcs atop the buildings. There are far too many arcs to be helpful in understanding the software architecture. The visualization of the same software with our layout in Fig. 1b is much clearer as it takes both the code structure and the directed dependencies into account. We organize buildings in layers. Most of the dependencies are implicit from one layer to the next. We only draw a dependency as an explicit arc if its orientation is opposing the order of the layering. These arcs often indicate architecture violations. Note that the TreeMap visualization uses simple buildings of the same sizes and colors, while in addition to showing only a few (problematic) arcs our visualization also makes architecture violations even more clear by mapping metrics to building properties, see "Creating City Artefacts".

Also commercial tools like Structure101 [7, 8] suffer from the lack of clarity. Figure 1c visualizes the Spring framework with Structure101. The layout is described in "Related Work" in more detail. In contrast to our layered 3D city representation, Structure101 is restricted because (a) zooming in/out and moving on the *x*-/*y*-axes are the only ways to navigate, while we offer arbitrary angles, (b) the nodes do not carry other information while we map metrics to the width, height, depth, and color of the city artifacts to promote a better understanding of the system. Our red buildings in Fig. 1b show dependency cycles more prominently. For instance, note the tall red tower in the violet circle that tells the software architect at a glance that this component is heavily used and hence to urgently refactor the cyclic dependency. In contrast, the same arc in the violet is easy to miss in Fig. 1c. Also, our red coloring of packages (orange cycles) indicates easier-to-resolve internal dependency cycles. Structure101 does not provide any such highlighting.

To sum it up, our main contribution is a layout that is based on layered graph drawing. The proximity of components in our layout correlates with both the dependency structure and the hierarchical source code structure. By encoding most dependencies in the layers instead of drawing them as explicit arcs, we significantly reduce their numbers and increase the overall clarity of the Software City. Arcs that are shown explicitly, often indicate architecture violations. To our knowledge we are the first to apply layered graph drawing to a 3D layouting of a Software City.

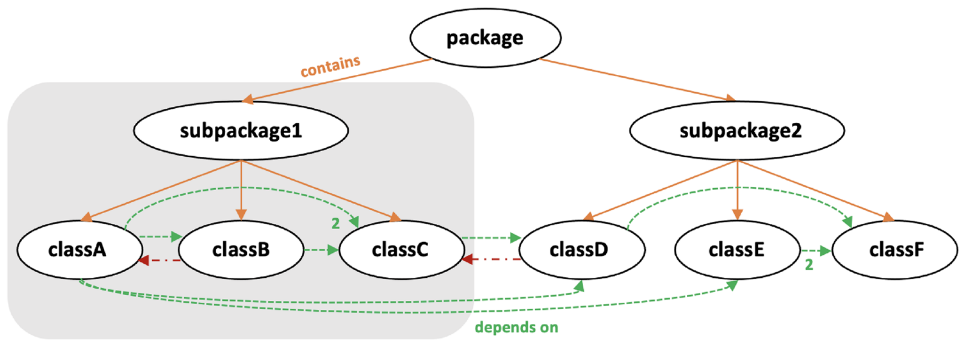
This article extends our work presented at the IVAPP 2021 conference [9] and also visualizes dynamic dependencies in the Layered Software City. As showing all dynamic dependencies as arcs would again clutter the visualization, a clustering technique finds use case clusters in the trace data and then only displays one representative per cluster. To further clean out the view, there are also filters both for the dependencies from/to a particular component and for single use cases.

This article is organized as follows: "Static Dependencies in a Layered Software City" explains the Layered Software City in detail, followed by a quantitative evaluation in "Evaluation of the Layered Software City". "Visualization of Dynamic Dependencies" covers and evaluates the visualization of dynamic dependencies. "Related Work" discusses related work before we conclude in "Conclusion".

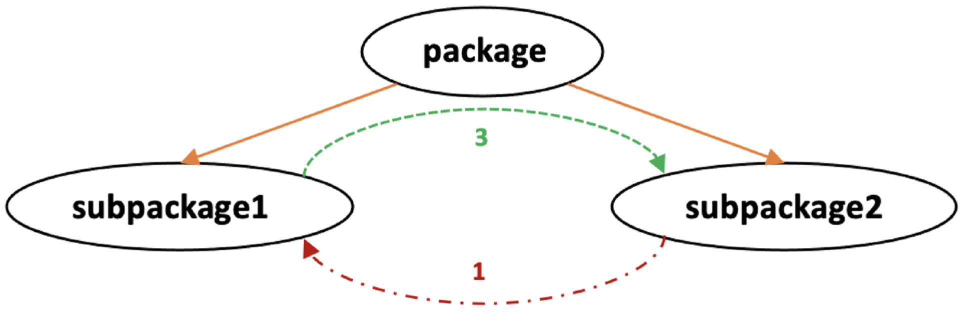
Static Dependencies in a Layered Software City

We propose to not only use the hierarchical contains relationships among elements of the static source code to build the layout of the Software City but to organize the City in levels that also reflect the depends-on relationships between artifacts. To achieve this, we propose to arrange the components on levels. As the components on one level in general

[1] https://spring.io/projects/spring-framework.



(a) Fine-grained graph with cycles.



(b) Coarsened view of the dependencies with cycles.

Fig. 2 Example of contains and depends-on relationships with cyclic dependencies

depend on the level below, these dependencies no longer need to be shown explicitly. Only dependencies in the other direction form cycles that are often problematic and should be avoided in well-designed software. To retain the static structure of the source code in the layout, the organization of city artifacts into layers is a recursive process, starting from the lowest level of detail (for example, class level).

The main steps for constructing a Layered Software City are:

1. Import raw contains and depends-on relations from the static source code. As in general the proposed layout is applicable to arbitrary graphs with two types of relations, we skip here how our implementation obtains the graph from a given code base. For details see "Evaluation of the Layered Software City".
2. Determine the level of each component and identify cyclic dependencies, see "Determining the Level" for details.
3. Create city artifacts for the components and position them based on their levels, see "Creating City Artefacts".
4. Draw arcs for identified cyclic dependencies.

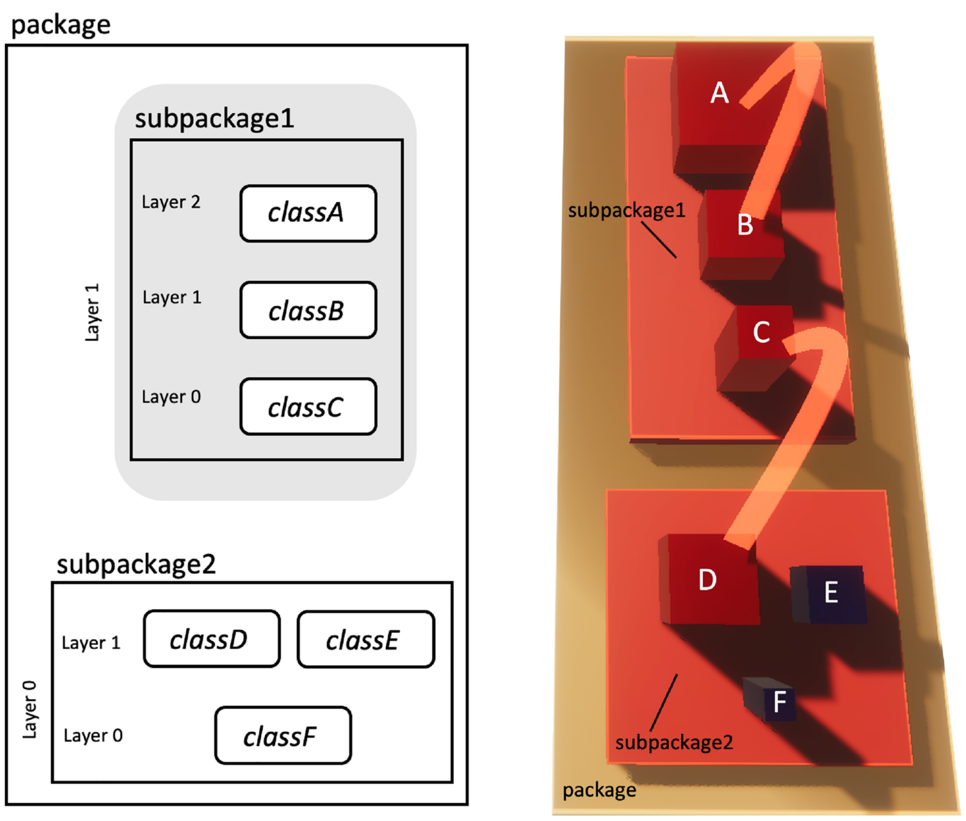
The two steps Crossing Minimization and Horizontal Coordinate Assignment of standard layered graph drawing [10] are not necessary for our layout because (a) we do not show most of the edges (arcs) in our visualization and (b) we consider the hierarchical structure of the software components, i.e., the nodes, which already ensures a certain proximity of nodes in our layout.

Determining the Level

In graph terminology we determine the layer of a node. We search for a layered drawing of a directed graph with two types of directed edges: structural contains edges and (weighted) *depends-on* edges. Let us assume an example dependency graph as in Fig. 2a with its structure edges (orange) and its dependency edges (dashed green with weights). The dependency edges in red are those we visualize explicitly. Let us postpone how we identify such cycle-building edges. We omit them when we present the basic idea of the layering algorithm. The hierarchical structure graph (orange edges only) is a rooted tree. Dependencies only exist between leaf nodes of the hierarchical structure.

A graph with many dependency edges is cluttered, confusing, and not very helpful for understanding the software. In the example, it needs a close look to see that subpackage2 is providing basic components to the rest of the system. What a software architect is mainly interested in when analyzing the code, are both the dependencies within a (sub) package and also the dependencies between (sub)packages, i.e., the dependencies per and among levels of abstraction.

Fig. 3 Dependencies encoded in the layering



(a) Abstract view. (b) Full visualization.

Basic Layering

We construct such a layout recursively. The base case is the layouting of those leaf nodes of the structural hierarchy that have a common parent. Here, we chose a layout that is inspired by a topological sort of the depends-on relations between those leaf nodes. We discuss the details below.

To make the dependency structure much clearer for the architect the recursive case one level up coarsens the graph as shown in Fig. 2b. Dependency edges between leaf nodes of the structural hierarchy turn into dependency edges between their respective parents in the structural tree. Resulting self-loops are dropped (3 times for subpackage1 and two times for subpackage2 in the example). Parallel dependency edges are fused and their count is kept as the weight of the fused edge. In Fig. 2b the fused depends-on edge has weight 3. The weights later become relevant when there are cyclic dependencies. Since the coarsened graph again has all its depends-on edges only between its leaves, we apply the same layouting inspired by topological sort. The recursion terminates at the source(s).

Let us now discuss the base case. We sort all the n leaf nodes that have a common parent in the structural hierarchy in a topological way. For the base case, only the e depends-on edges among the set n matter. We ignore dependency edges that come into this set from other leaf nodes or that leave the set. Whereas a text-book topological sort has room for variation, we determine the unique layer of a node as its maximal path length from it to the last node among the set n, that has no more outgoing dependencies. Listing 1 holds the pseudo code. Its complexity is $O(|n| + e)$ with e depends-on edges among the nodes in n.

Consider the shaded area in Fig. 2a. As classC has no outgoing depends-on edges that stay within the set n, its layer is 0. There are two paths from classA to classC. As the longest one has length 2, this is the layer of classA.

Once the node layers in n are computed, we draw them layer-by-layer, leaving out depends-on edges from layer i to j when $i < j$. We draw items on the same layer in a random order. In the example, the three class nodes of the shaded area in Fig. 2a turn into the three layers in the shaded area in Fig. 3a (in 2D for simplicity). As the layouting process is recursive, the layers determined for the coarsened graph in Fig. 2b result in the shown layering of the subpackages in Fig. 3a.

In this visualization the architect can easily identify that items depend on the items below them. The structural hierarchy is also still present. Note that while the abstract graph in Fig. 3a ignores the cyclic arcs, they are already present in the full visualization in Fig. 3b. "Creating City Artefacts" discusses the building properties.

In our layouting task a given graph has a total of N nodes, each of which has at most one parent, i.e., it is in a set n only once. In the recursive procedure described above each of the total E depends-on edges is considered only once, in one of the sets e. The total asymptotic complexity of the layering is hence $O(N + E)$.

```
function topoLayout(nodes n):
  degree[] = [Out degrees of n]
  q = {set of all dependency leaves}
  current = 0
  while q is not empty:
    qNew = {}
    for each node k ∈ q:
      k.layer = current
      for each incoming edge s→k:
        degree[s]−−
        if degree[s] == 0:
          qNew ∪= {s}
      q = qNew
    current++
```

Listing 1: Layouting inspired by topological sort.

Dealing with Cycles

If there are cyclic dependencies, there are edges whose directions do not fit the layering and thus need to be visualized, see the cycle-building arcs in Fig. 3b. The fewer arrows a drawing has and the shorter they are, the easier to understand the visualization is. In the example, it is apparent that the dependency from classB to classA as well as that from classD to classC need refactoring.

While in general finding this ideal visualization boils down to the NP-hard Minimum Feedback Arc Set problem [11], for software systems we can give domain specific heuristics that usually work well. The underlying assumption is that a software system is not utterly broken and that the majority of the dependencies fits to the layered software architecture, i.e., first that cyclic dependencies are rare and second that they are mostly quite local and do not affect software artifacts that are "far away" in the code, either syntactically or semantically. The reason is that such issues are architecture violations that developers have learnt to avoid and because it is a common refactoring task to remove them. In the layered drawing we do not show dependencies that fit the general layering of the software architecture, i.e., its major direction. The (feedback) arrows of the few dependencies that have the opposite direction and that close cycles, highlight potential architecture problems. If in a broken software architecture there is no identifiable flow of dependencies in one major direction, i.e., if there is no class layering of the software architecture, a detailed analysis of the source code must be performed anyway. In such cases it does not really matter which of the cycle-building edges is highlighted by means of an explicit arrow.

The remainder of this subsection discusses in detail how we identify the (few and short-range) cycle-building edges that need to be visualized.

As suggested by Sugiyama et al. [10], a pre-processing in each of the above recursive steps already removes cycles from the graph. Once a depends-on cycle among the leaf nodes of the structural hierarchy that have a common parent is detected in a depth-first traversal, we immediately remove one of the cycle-building edges. The acyclic rest of the graph can be drawn in layers and without arrows as before. The removed edge is later added as an arrow atop those layers. Although conceptually a depends-on edge may belong to several cycles, we have not seen such a case in practice. Dependency cycles often seem to be disjunct in real software. Even if the cycles are not disjunct, we argue that the cycles still belong to disjunct use cases. They have most likely been implemented at different times and solved different tasks. Thus, from a software architect's point of view, there is no need to find the absolute global optimum when minimizing the number of arrows. So in addition to having only few cycles and only relatively short ones in our graphs, edges in general only belong to at most one cycle. That makes the visualization much simpler than solving the general Minimum Feedback Arc Set problem: with linear asymptotic complexity, we can simply traverse the graph. Once we detect a cycle in this traversal, we remove it instantly as soon as we found it. While there are E depends-on edges in the given graph, the cycle only has $e << E$ edges.

We use two heuristics to pick which of the e edges of a cycle to remove. Heuristics 1: If there are two edges in a cycle and one edge has a higher weight, the higher weight indicates the layering that originally was planned for the software system. Hence, we pick the edge with the minimal weight for removal. In the cycle in Fig. 2b it is obvious that subpackage1 is meant to depend on subpackage2 and that the red edge is the dependency that needs to be refactored, i.e., that should turn into an explicit arrow. Heuristics 2 comes in if the cycle has more than one edge with minimal weight. Let's call these edges removal candidates. To motivate the heuristics, consider the shaded area in Fig. 2a. There are two ways to resolve the cycle between classA and classB, as the weight of the edges is the same. Removing the (red) edge from classB to classA means for the layering that classA is placed above classB. Removing the (green) edge from classA to classB results to classB above classA. Which one is better for the software architect? A common design principle of software is that the more complex component uses the less complex one. Heuristics 2 uses the correlation between complexity and the number/weight of outgoing dependencies of a component [12]. In Fig. 2a, the sum of the weights of the outgoing edges of classA is 3, whereas for classB it is only 2. As classA is more complex, the software architect expects the visualization in Fig. 3.

Hence, heuristics 2 is: if there are two removal candidate edges (with the same minimal weight) the one whose source node has a larger total outgoing weight reflects what was planned to be the node that makes use of others in the software system, as it represents the more complex component. So we pick the candidate edge for removal whose source node has the minimal total outgoing weight.

```
function removeCycle(edges e):
    minWeight = MaxInt
    removalCands1 = {}
    // heuristics 1
    for each edge k ∈ e:
        if k.weight < minWeight:
            minWeight = k.weight
            removalCands1 = {k}
        elsif k.weight == minWeight:
            removalCands1 ∪= {k}
    if |removalCands1| == 1:
        remove edge ∈ removalCands1
    else:
        minWeight = MaxInt
        removalCands2 = {}
        // heuristics 2
        for each $e = (s_e \rightarrow t_e)$ ∈ removalCands1:
            out = {set of all outgoing
                     edges of $s_e$}
            weight = $\sum_{o \in out}$ o.weight
            if weight < minWeight:
                minWeight = weight
                removalCands2 = {e}
            elsif weight == minWeight:
                removalCands2 ∪= {e}
        if |removalCands2| == 1:
            remove edge ∈ removalCands2
        else:
            remove random edge ∈
            removalCands2
```

Listing 2: Cycle removal heuristics.

Listing 2 shows the pseudo code of the two-stage heuristics for removing cycles. It first traverses the edges e that form a cycle of length $|e|$ to find the ones with minimal weights. For the cycle in Fig. 2b this traversal finds only one candidate for removal. Heuristics 2 is not needed in this example. For subpackage1 there are two candidates of minimal weight 1. In the worst case all $|e|$ edges have the same minimal weight, i.e., the traversal for heuristics 2 takes another $O(|e|)$. After removing an edge, $|e| - 1$ edges remain that may conceptually be part of other cycles. This leads to a worst-case complexity of $O(|e|^2)$ per cycle of length e. For general graphs with potentially many long cycles the complexity would add up to $O(|E|^2)$ per each of cycles say i.

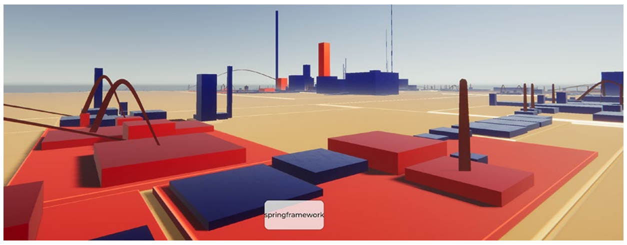


Fig. 4 Zoomed-in view of the Spring core Layered Software City

This is much worse than what general purpose heuristics for the Minimum Feedback Arc Set problem achieve. But since our special case only has few, say i, cycles and each cycle is short, and each edge belongs to one cycle at most, for the overall complexity we have $O(\sum e_i^2) << O(i \cdot |E|^2)$. In practice we see cycle lengths $e_i \leq 3$, resulting in a constant upper bound per cycle, for a small number of cycles. As a result, the `removeCycles` function rarely needs to be called, and if so, then only with two to three edges as parameters from which one must be selected for removal. In practice the cycle removal takes only a few hundred milliseconds even for large software projects. For example, it took 90 ms for the 19.732 dependencies of the benchmark project, see "Evaluation of the Layered Software Citylayout", on an Intel Core i7 laptop.

The layout is stable as long as the cyclic dependencies do not change. If the dependencies in the system change, a class may be assigned to a different level than before. Since such a change probably indicates an unwanted modification of the software architecture, it is useful to see this in a shift of the layers.

Creating City Artifacts

What is left after having determined on which level to put a component is how to employ other visual properties of its building or district. We use dependency metrics for this purpose since we aim at visualizing dependencies in software.

We set the height of a building based on the number of incoming dependencies. The square area (width = depth) reflects the outgoing dependencies. Tall towers describe classes that are used a lot, while flat buildings with a wide footprint visualize classes that use many other components. The color of a building indicates whether or not the class belongs to a cyclic dependency. We use a red color to mark cycles on buildings as well as on the districts. We display the identified edges of cycle-building dependencies with explicit arcs between the components. The arcs show the dependencies that probably represent architectural violations. When a user clicks on a component (building or district), we display its name.

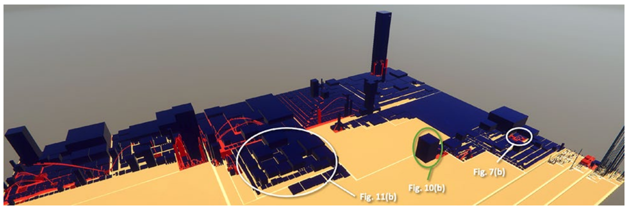


Fig. 5 Software City with TreeMap layout of SolrJ

Figure 4 shows a zoomed-in view of the Spring core visualization, where the described city properties can be seen in detail. Buildings of components that form a cycle are colored in red, as well as the corresponding districts. The cyclic dependencies appear as arcs between the respective buildings. The tall buildings in the background are examples of heavily used components, while the flat buildings in the foreground represent components that use many others.

Evaluation of the Layered Software City Layout

In order to answer the research question "Does the layered layout for Software Cities help to better understand the architecture of a software project", we evaluated our approach in a controlled experiment. In the study we used a real-world software system, i.e., version 9.0.0 of the open source project SolrJ,[2] which is the Java API for Apache Solr, a standalone enterprise search server for any kind of documents. SolrJ has 177 740 lines of code in 974 classes/interfaces. We analyzed the dependencies of the SolrJ jar file with the command line tool `jdeps` that is included in JavaSE since version 8. It analyzes all static dependencies between Java class files and stores the depends-on relations in a text file. We derived the contains relations from the class name, e.g., `package.subpackage.class`. This resulted in 19 732 depends-on relations. We created the Software City of SolrJ, once with the most common TreeMap layout (see Fig. 5) and once with our new layout (see Fig. 6). In the study, the participants used the Software City visualization to find answers for a set of questions within a given time limit.

Note that a standard TreeMap Layout would show all dependencies as illustrated in Fig. 1a. Figure 7a shows how the spot in the white circle on the right of Fig. 5 would have looked like if we would show all dependency arcs. In a prestudy participants were overwhelmed with the many dependency arcs and were unable to answer any questions at all. Therefore, we improved the TreeMap layout— like in our layout—by only showing the architecture violation arcs plus the affected classes and with the dependency metrics used to determine the visual properties of their buildings. Figure 7b illustrates the effect: cyclic dependencies and affected classes are easier to see in the enhanced TreeMap Layout.

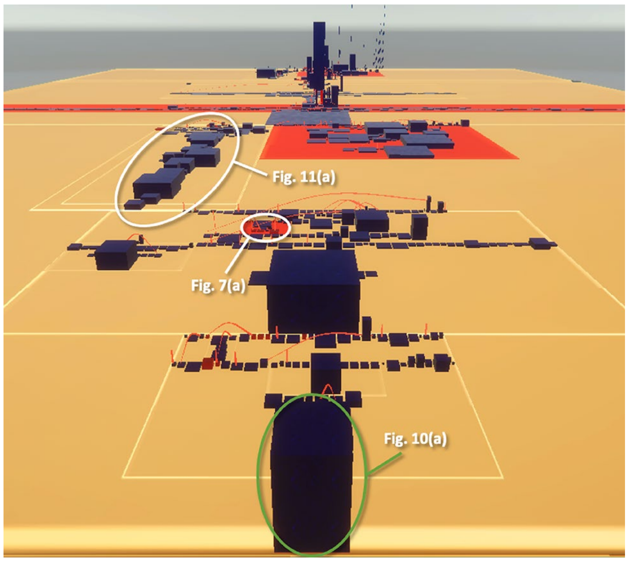


Fig. 6 Layered Software City of SolrJ

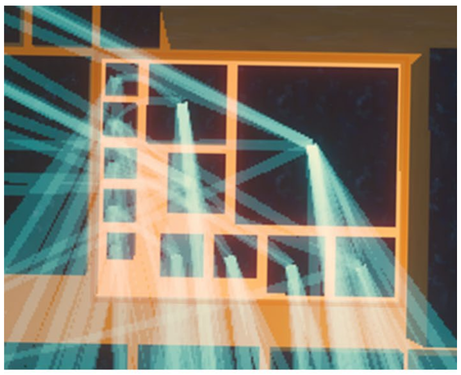
Since the properties of the buildings and districts (height, width, depth, color) were identical in both the enhanced TreeMap Layout and in our layered layout and as also the representation of cyclic dependencies with arcs was the same for both cities, the only difference was the layout.

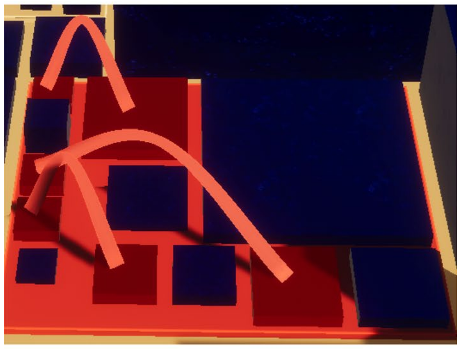
Participants

A total of 30 professional software engineers of QAware conducted the study during their working hours. They all have a computer science or similar background and are familiar with concepts such as software architecture, dependencies, and cycles. The company provided the resources because they are looking for a visualization that their employees can use to get productive in newly assigned projects more quickly. The supervisor knew the participants from work. The professional experience of the test persons ranged from 1 month to over 10 years. The majority of the test persons have a work experience of 2–5 years (43.3%), see Fig. 8.

We randomly assigned 15 participants to the Enhanced TreeMap group and 15 to the group that uses the Layered Software City layout. We had only two female participants, one in each group. None of the participants had used a Software City visualization before. The participants only knew that the purpose of the study was to pick among two layouts. All participants were informed that they would solve tasks and that both the answers and the response times would be

[2] https://lucene.apache.org/solr/.


(a) TreeMap layout with all dependencies shown.


(b) Enhanced TreeMap Layout with only cycle-building edges and our building properties.

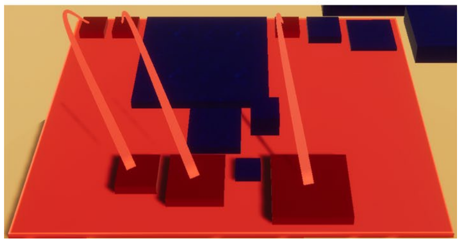

(c) Cycles in our layered layout.

Fig. 7 Zoomed-in view of a spot in Figs. 5 and 6 that is relevant for task #4

documented. They were also told that they would have to fill out a questionnaire afterwards.

Experiment

The experiment was performed remotely. The participants received executable files of the visualizations in advance. The study itself was then conducted via video conference and screen sharing. To warm up, all test persons initially received a playground project, which was also created with the layout of their respective group. The participants had five minutes to get familiar with the navigation. During this time, the supervisor used a script to explain the visual properties

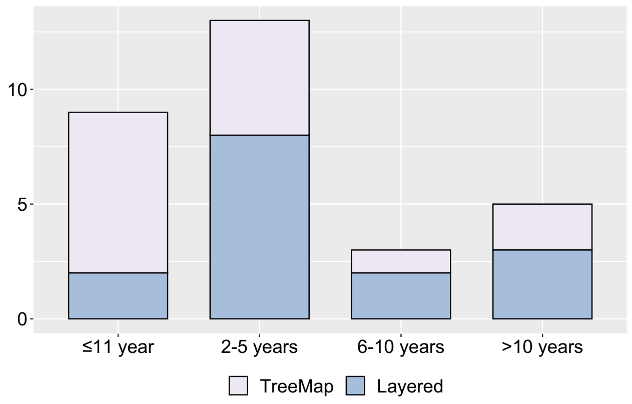


Fig. 8 Years of work experience of the participants for the TreeMap group in dark blue on top and for the Layered group in light blue below

(height, footprint, color) as well as the layout. Participants were allowed to ask questions.

After this familiarization phase, the SolrJ study started. The participants had to solve 7 tasks in which they had to analyze the software architecture:

1. Which class is the entry point in SolrJ? (2 min)
2. Locate package “util”. (2 min)
3. Locate package “impl”. (2 min)
4. Which dependency would you refactor in a 1–1 cycle of your choice? (2 min)
5. Find the component in the system that is used most. (1 min)
6. Find a package with a deep dependency tree and one with a flat one. (4 min)
7. Specify how you would refactor all cycles in package “noggit”. (4 min)

The tasks were tailored to the software system that the participants were supposed to analyze. Nevertheless, we asked questions aimed at skills that are generally required for software analysis. Tasks #1 to #3 reveal how well and quickly participants can orientate within the visualization. Tasks #4 and #7 expose whether the visualization supports refactoring issues. And tasks #5 and #6 target the question of how well a layout can give a broad overview of the architecture.

The tasks were posed one after the other. The supervisor did not give any feedback on the correctness of the answers and hence on the subject’s comprehension of the software architecture. The upper part of Fig. 9 holds the results.

There was a maximal response time per task, given in parentheses above. Time measurements started once a task was posed. If no answer was given within the allotted time, the answer was considered incorrect and the time limit was noted with an added *fail* mark (the red x with the number

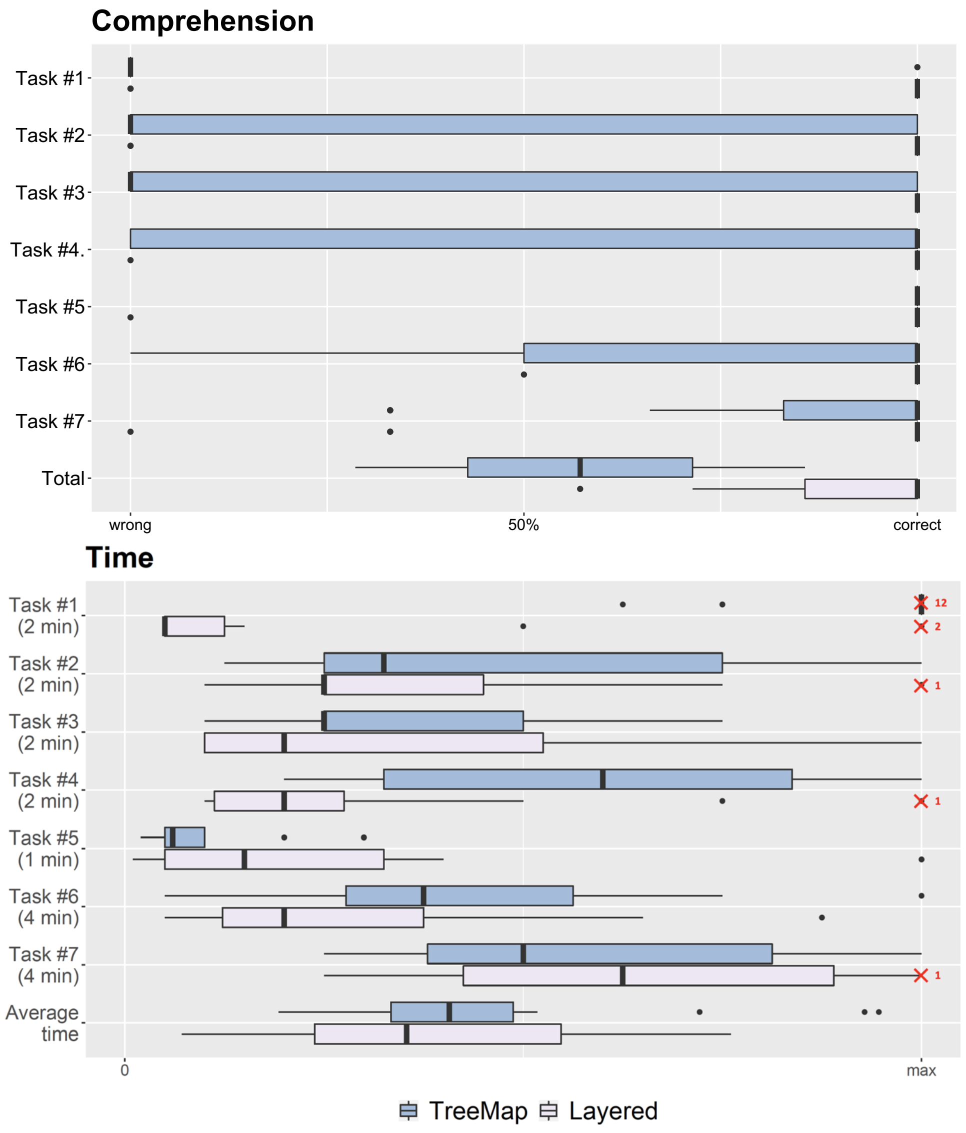


Fig. 9 Comprehension (top) and Time (bottom). For each task, the distribution of answers for the TreeMap group in dark blue on top and for the Layered group in light blue below. Boxes correspond to the first and third quartiles (the 25th and 75th percentiles), whiskers drawn using Tukey method (1.5 IQR), points are outliers in the data. Failures to solve a task due to the time limit are shown with a red x and the number of such failures

of failed answers in the lower part of Fig. 9). Otherwise, the time required was documented. The time limits per task suited the complexity of the question and were determined in a small preliminary study. After the 5 minutes warmup the maximal duration of the experiment was 20 minutes. Most participants finished sooner.

After all tasks had been completed, the participants had to fill out an anonymous questionnaire asking for general information, e.g., years of professional experience or position. In addition, we used the standardized NASA Task Load Index (TLX) to make a comparable statement about the effectiveness of the two visualizations [13]. We added this questionnaire to be able to make a more general statement about the effectiveness of the visualization besides the specific task solving. In the same style, participants were asked about how much the layout helped them in solving the tasks. There was also a free text field for further comments.

Results and Discussion

Comprehension: The total results in Fig. 9 (top) show that the Layered group solved the tasks significantly more correctly than the TreeMap group (significance level $\alpha = 0.01\%$ determined by a Chi-squared test). In total, the Layered group reached a median correctness of 100% with the first and third quartiles spreading from 86 to 100% and one outlier outside the lower whisker at 56%. In contrast, the TreeMap group only achieved a median of 57% correct answers (quartiles spread from 37–69%). The detailed results for the seven tasks vary. As the Layered group was almost always correct, we show only the medians and outliers (no boxes, no whiskers).

Task #1 has been solved correctly by 13 participants of the Layered group but only by one of the TreeMap group. The layered layout is ideal for this task as it arranges the

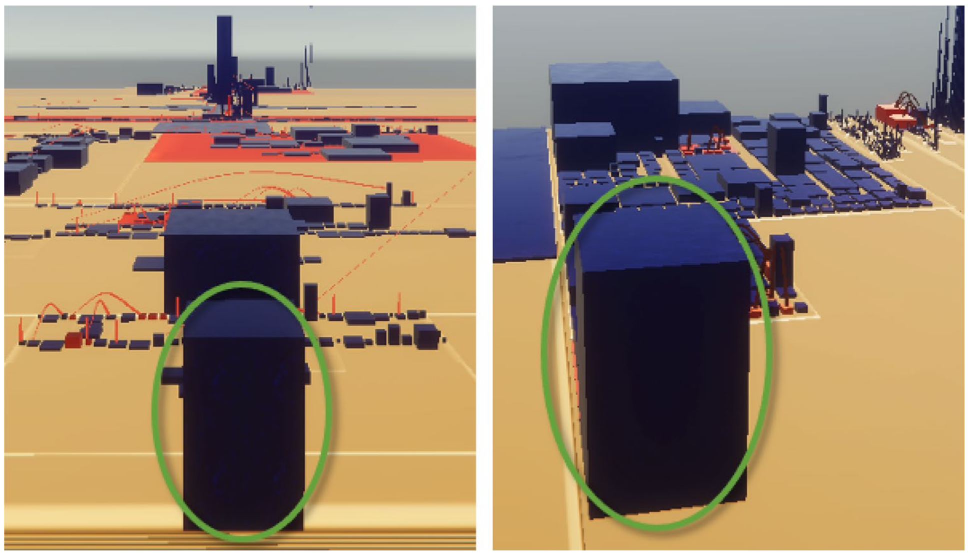


Fig. 10 Helpful viewing angles to solve task #1. The green circles can also be found in Figs. 5 and 6

(a) Layered: view from the top layer. (b) TreeMap: view from best corner.

components according to their dependencies so that the entry point is in the foreground when users view the city with a perspective from the top layer. We added a green circle to highlight this in Fig. 10. The TreeMap layout places the items solely based on their footprint sizes. There is no way to guess the entry point. Even when we pick the best possible angle to view the city in Fig. 10b, this view still does not reveal the dependencies. Note that for orientation, the green circles are also present in Figs. 5 and 6 .

For tasks #2 and #3 the Layered group also performed better. In the layered layout the "util" package, which contains all auxiliary classes of SolrJ, is used by many and is therefore further down in the layering. The "impl" package, which contains the implementation of business logic and uses many components, is thus shown further up. This helped the Layered group in finding the respective packages. There is no such help in the TreeMap layout.

Only for task #5 (identifying hotspots) both layouts score equally well. The TreeMap group is better than usual as the layout is compact and hotspots can be easily recognized. But it also shows that hotspots are not less visible with the layered layout, so the more extensive layout does not have any disadvantage.

Time: We also measured how long it took the participants to solve the tasks. If they exceeded the maximal allotted time, the task was also judged non-solved. Figure 9 (bottom) shows the results of the time measurements. The time interval is normalized to an interval from 0 to the time limit. An exceeding of the time limit is marked as a separate data point to the right of the maximum.

The average responding time for the TreeMap group is 40.7% (median) of the time limit while it is a better 35.4% for the Layered group. We do not consider the correctness of the answers here. Overall, the Layered group solved the tasks not only qualitatively better, but also significantly faster (significance level $\alpha = 0.01\%$ determined by a Chi-squared test).

The TreeMap group detected hotspots (#5) faster since the layout is more compact (see above). The TreeMap group was also faster with task #7, but there were also more wrong answers while the median in the Layered group was correct.

The time difference in task #4 is also worth explaining. The layered layout arranges buildings so that architecture violations in cyclic dependencies are displayed as arcs from lower to upper layers. Fig. 7c zooms to such a spot in the SolrJ visualization in Fig. 6. The Layered group easily spotted such patterns. In contrast, the TreeMap group had to derive the dependencies and the resulting complexity solely based on the building properties (height and footprint, see Fig. 7b) since the arcs have an irregular pattern. This took longer, even though we did not present all dependencies which is the standard in Software Cities with a TreeMap layout (see Fig 7a) but used our color encoding of the affected class buildings and only showed the architecture violating arcs to the TreeMap group.

There is a notable time difference for task #6. In a bird's eye view, the layered layout instantly reveals which packages have a deep tree of dependencies. Figure 11a zooms into one of the packages of the SolrJ visualization in Fig. 6. A similar view does not help the TreeMap group at all (Fig. 11b).

Questionnaire (NASA-TLX): We also used the NASA Task Load Index (TLX) questionnaire [13] to measure the effectiveness of our visualization and to compare it to the TreeMap layouts in a standardized way. As the weighting of the six dimensions originally proposed by the authors has been criticized [14], we made an unweighted evaluation

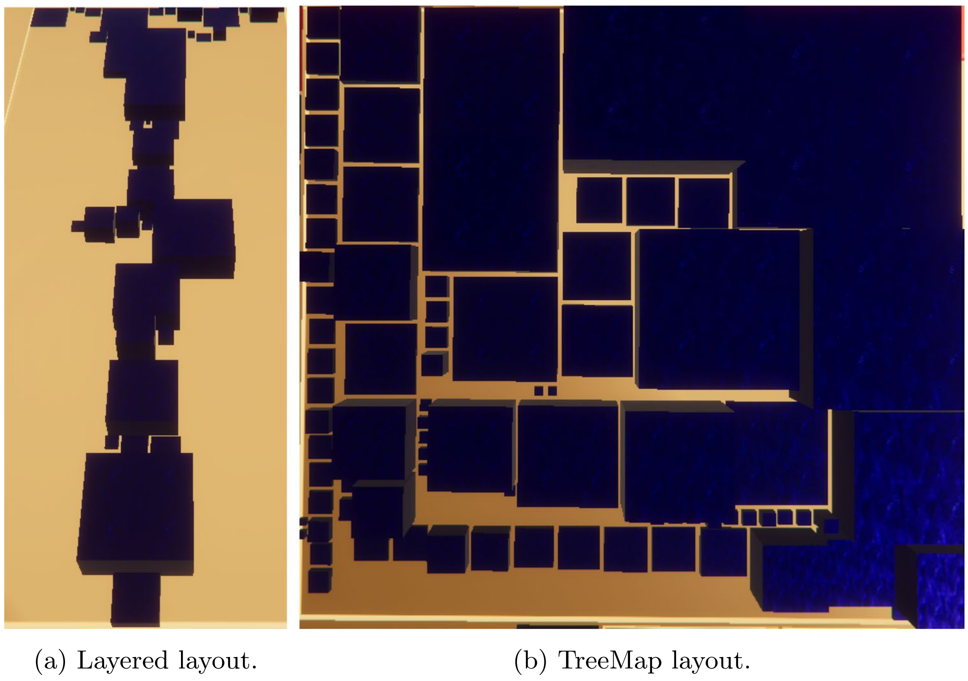


Fig. 11 Zoomed-in view of a package with a deep tree of dependencies needed to solve task #6. Same areas as in Figs. 5 and 6

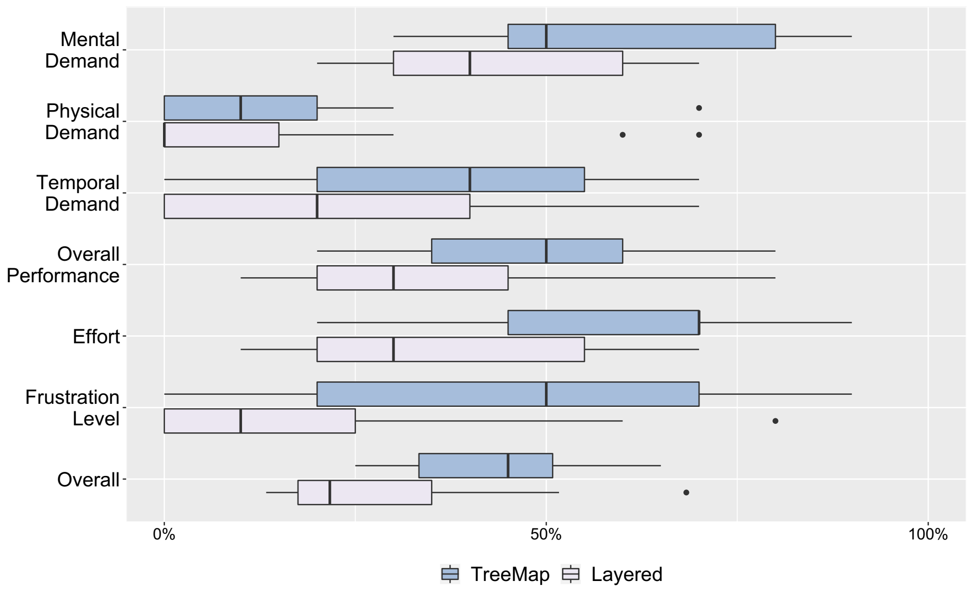


Fig. 12 Questionnaire (NASA-TLX). Evaluation of the task load. For each question, the distribution of load for the TreeMap group in dark blue on top of the Layered group in light blue below

according to the latest recommendations. Figure 12 shows both the comparison of the two layouts in each dimension and the overall Task Load Index. It is obvious that the cognitive load for solving the tasks (all but the physical demand dimension) is lower for the Layered group than for the TreeMap group. The median of the overall Task Load Index is 45% for the TreeMap group compared to significantly lower and better 22% for the Layered group (significance level $\alpha = 0.01\%$ determined by a Chi-squared test). The participants of the Layered group did not show any signs of cognitive overload.

We added an extra summary question to the questionnaire: “How much did the layout help you in solving the tasks?” As can be seen in Fig. 13, there is again a significant difference between the two layouts (significance level $\alpha = 0.01\%$ determined by a Chi-squared test). The TreeMap group found the layout in 40% (median) supportive, but 70% of the Layered group indicated the layout helpful.

There is only indirect evidence about the handling of cycles. Many participants correctly and more quickly solved the refactoring tasks that have to do with cycles (#4 and #7 in Fig. 9). In their answers they often referred to the

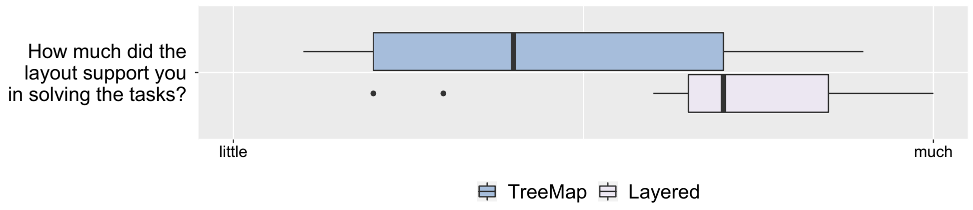


Fig. 13 "How much did the layout help you in solving the tasks?"

arrows (e.g., "I would refactor the upwards arrow"), while the TreeMap group solely used the building properties and names in their responses.

Also the free text fields varied a lot between the two layouts. In the questionnaires of the TreeMap group we encountered the following word heaps at least twice: "no help", "not intuitive", "difficult to find the right conclusions". Those word heaps did not occur in the Layered group. In contrast, there we found word heaps like "supported", "quick to recognize", "intuitive", and "easy to use".

In conclusion, the study has shown that without the visual clutter of too many arrows and with the layering according to the main direction of dependencies, our layout makes it easy to intuitively understand dependencies between components. The test persons also appreciated the handling of cycles and considered the resulting arcs to be helpful in the refactoring task. The layered layout of the Software City can be used to analyze software architecture and outperforms the default TreeMap layout, even in its enhanced version. The layered layout can also keep up with the typical use cases of the TreeMap layout like detecting hotspots. Most participants stated that the layout supported them strongly in solving the tasks.

Threats to Validity

We assess the threats to validity of our study as low. Although we randomly assigned the participants to one of the two study groups, we only discovered afterwards that the Layered group on average had 1.5 years more professional experience, see Fig. 8. This imbalance is a potential threat as the fraction of participants with longer work experience (1/3 vs. 1/5) may have caused the differences in the results. To gauge the impact of the fraction of seniors, we re-ran the analysis with the data only of the less experienced participants (<6 years). The overall correctness of solving the tasks for the TreeMap group got worse (from 57 to 50%), while it remained the same for the Layered group (100%). This still is statistically significant despite the smaller group sizes. Therefore, the slightly higher seniority of the Layered group was not the cause of its better performance.

Since our layout is primarily designed for the visualization and analysis of dependencies, we chose the tasks in the study accordingly. When designing the tasks we made sure to check required skills such as orientation, refactoring, and clarity. If the participants had to solve tasks, such as quickly finding the component with the largest area, the more compact TreeMap layout would probably score better. For our study, however, the focus was on the analysis of dependencies, and for this purpose we set the tasks so that they surveyed generally important skills.

As male and female participants were equally distributed in the two groups, gender specifics did not skew the results. But one female participant per group is far from enough to conclude that the results hold for all genders.

Another potential threat is that the participants knew the supervisor and have guessed that the layered layout should perform better. As countermeasures, all participants were encouraged to solve the tasks as best as possible. Furthermore, the TreeMap group scored better on task #5, which would not have been the case if participants had tried to influence the outcome of the study.

As we fixed all other parameters of the visualization, such as colors, building heights, etc., and only changed the layout, non-layout differences did not influence the results. We even highlighted architecture violations and cyclic dependencies to help the TreeMap participants find spots of interests.

Visualization of Dynamic Dependencies

To view the dynamic dependencies, the user can switch the Software City to a night view that leaves the layered layout unchanged, but instead of cyclic dependencies displays dynamic dependencies as arcs. We use a neon color scheme as it mimics currently popular computer games. As software developers are the users of our visualization and they often also play computer games, the neon color scheme appeals to them more. A design company also encouraged us to choose neon colors. We also wanted to have a clear separation between the visualization of static and cyclic dependencies for analyzing the software architecture and the visualization of dynamic dependencies. Thereby, the day/night view fits well into the city metaphor.

Figures 1a and 7a already demonstrate that displaying all dependencies, whether static or dynamic, leads to a cluttered view. To reduce the number of arcs when visualizing dynamics this section presents a novel trace clustering that groups traces according to similar call structures, i.e., use cases, and only shows one representative trace per cluster instead of

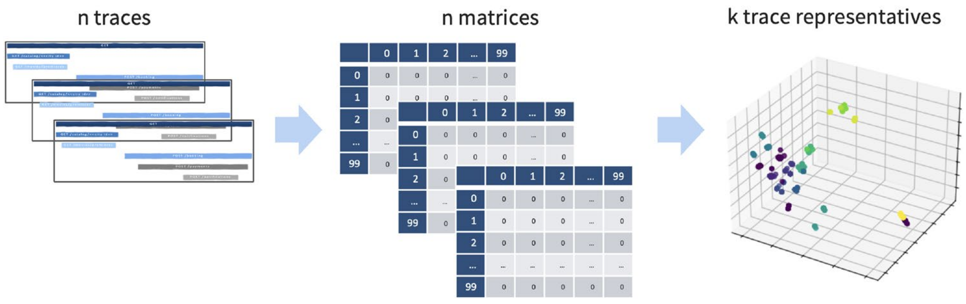


Fig. 14 Steps of the clustering process

all traces. This reduces the number of explicitly drawn arcs. Some additional filter options further clean out the view.

Trace Clustering

Trace Representation: Assume a software system with N components/endpoints. To recognize call patterns we represent each of the n traces from the runtime data as a separate $N \times N$ matrix. As in real-world systems N is too large for reasonably sized matrices, we hash the names of the endpoints to keep the dimensions of the matrix bounded. With the hashing of the endpoint names we achieve two advantages: (a) the clustering is applicable to systems of any type and (b) changes such as addition/removal of endpoints can be modeled, since in both cases the dimension of the hash space remains constant. We use a hash table of size 101, a prime number to get an even distribution of hash values. For a system with over 101 endpoints this will cause collisions, i.e., two or more endpoints can have the same hash. However, in our clustering we do not look at individual endpoints, but always at sequences of endpoint calls, i.e., entire traces. In practice, trace lengths are at least 10 calls long; 25 or more calls per trace are not uncommon. The probability that all the hash values of all these individual calls collide and that different traces yield the same pattern is low. In the study example no such cases occurred. The matrix of a trace is initialized with zeros. If in the trace a component i calls a component j, we add the call duration to the matrix at position (i, j). As shown in Fig. 14 there is a matrix for each trace.

Clustering: We feed the matrices into a 3-component Principal Component Analysis and apply a DBSCAN clustering. This identifies k sets of similar call patterns that most likely semantically belong together and form a use case of the application. The plot on the right of Fig. 14 uses one color for all traces of a cluster. Developers can manually assign a name to a cluster that reflects its semantics in the application. In general, this is the name of the use case.

We picked this clustering algorithm because (a) we do not know the number of clusters in advance, (b) we potentially have a large number of samples, (c) the shape of the clusters can be diverse, and (d) we have noise in the trace data. We fine-tune the algorithm by adjusting the minimal number of samples per cluster and use a grid search with the silhouette score as a metrics to evaluate performance. The silhouette score is a textbook metrics for the quality of a clustering [15]. It indicates how close a point in a cluster is to points in neighboring clusters. The score is in [– 1,1], where – 1 means a false cluster assignment (worst), 0 means that a sample is close to the cluster boundaries, and 1 means that the sample is far away from the neighboring clusters (best). It is used in many clustering approaches to find a good-fitting clustering [16–18].

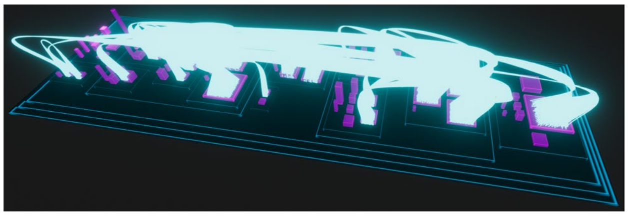
Cluster Representatives: We then use the point closest to a center of the cluster as the representative of the entire cluster. Showing only this representative in the Software City instead of all traces significantly reduces the clutter.

To better demonstrate our visualization of dynamic dependencies we have to switch the running example. Whereas in the SolrJ system used so far the traces mostly follow the static dependencies, the open source blogging project Spring Boot Realworld Example App[3] (with a modified microservice architecture) has more interesting trace data in load tests that ran for a total of about 15 minutes. Fig. 15a shows what a visualization of all traces would look like. The visualization is cluttered and it is difficult to follow the arcs to see which components communicate with each other. Even with a simple edge-bundling, the many arcs that must be displayed obscure the buildings behind them and additional navigation or perspective changes are needed to see the hidden components. In contrast, Fig. 15b only displays the trace representatives from the clustering, one per use case. The resulting fewer arcs clean out the visualization and it is possible to see all buildings. The dynamic dependencies are easier to understand and to follow.

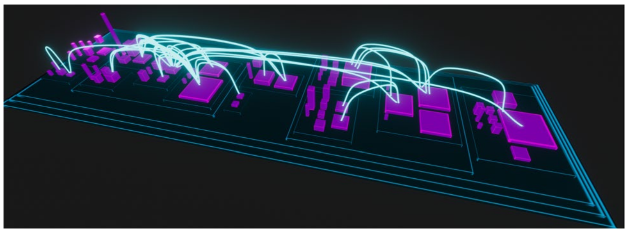
Further Filter Options

In addition, there are two filter options to further reduce the number of arcs.

[3] https://github.com/gothinkster/spring-boot-realworld-example-app.

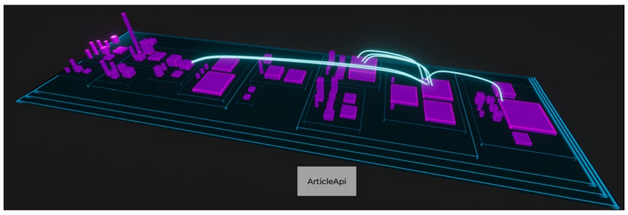


(a) Showing all dynamic dependencies as arcs.

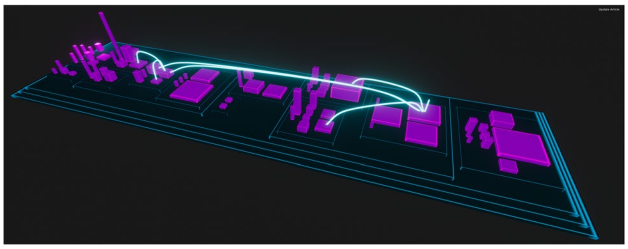


(b) Showing only the cluster representatives.

Fig. 15 Night view of the Spring Boot example



(a) Filter on dependencies from/to `ArticleApi`.



(b) Filter on the *Update Article* use case.

Fig. 16 Further filter options

Filter on components' dependencies: With a mouse click on a component, only the dependencies from/to this component are shown. In addition, the name of the component is displayed as in the day view. Figure 16a shows the effect of the filter on the `ArticleApi`. The filter reduces the number of arcs again and it is easy to see the dependencies of this component.

Filter on use cases: The user can filter on a particular use case using the arrow keys. If a name has been assigned to a trace cluster this use case name is displayed in the top right corner. Figure 16b shows the effect of the filter on the use case Update Article. The components involved in this use case and how they communicate with each other can be quickly identified.

Evaluation

To evaluate the usefulness of the clustering and the two filter options, we conducted a short study based on the blogging project and the above figures. We used the Elastic stack[4] to gather the trace data from the blogging project but every other tracing framework would also have worked.

Subjects: From the set of participants described in "Participants" a subset of 22 professional software developers were available for this study. They had a similar average work experience (2–5 years) and we again had 2 female subjects.

Experiment: The participants received a textual and illustrated description of both the visualization and the filter options. Also, the subjects were given the option to download the visualization and to try it out. None of the participants felt a need to do so, presumably because they all were familiar enough with the visualization and controls so that there was no need for an extra demo. In an online questionnaire they had to evaluate in which typical software engineering tasks both filtering options would help them, and how much. There was also a free-text field to comment on the ranking. The tasks were inspired by the knowledge areas defined in the IEEE Guide to the Software Engineering Body of Knowledge:[5]

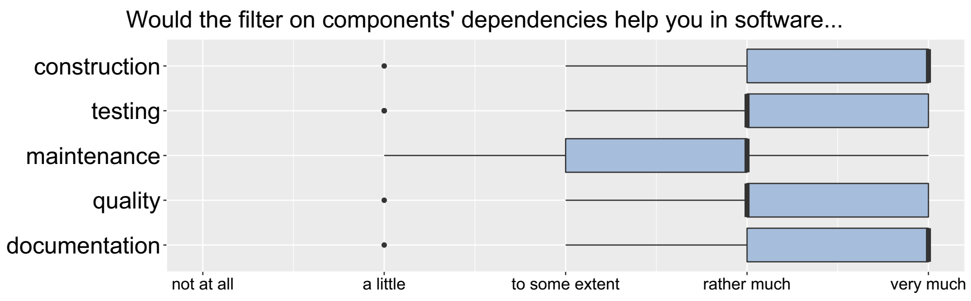
- Software construction, e.g., API design and use, coding.
- Software testing, e.g., unit or E2E tests.
- Software maintenance, e.g., software comprehension, refactoring, retirement.
- Software quality, e.g., architecture validation, reviews, audits.
- Software documentation, e.g., flow charts, UML diagrams.

Results and Discussion: Figure 17a shows the answers of the participants for the component dependency filter. The median of the participants ranked this filter as (much) helpful for all five software engineering areas. In particular, for software creation and documentation, the participants have considered the filter to be very helpful. In the free text field they stated that it is easy to directly see the dependencies of an external API and where it is used. They have also positively pointed out that when changing the code of existing components, one can see at a glance which other components depend on it and take care of not breaking any dependencies. These features are especially useful for software creation and documentation.

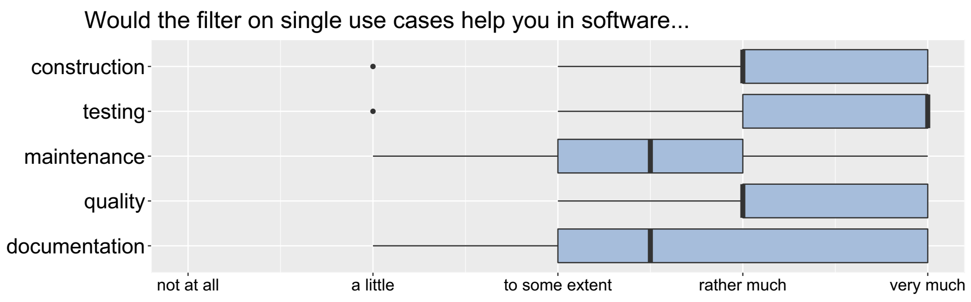
[4] https://www.elastic.co/elastic-stack/.

[5] https://www.computer.org/education/bodies-of-knowledge/software-engineering.

Fig. 17 Filter helpfulness for typical software engineering tasks



(a) Helpfulness of the component dependency filter.



(b) Helpfulness of the use case filter.

The subjects considered the use case filter to be less helpful than the component dependency filter, although, the differences were not statistically significant (except for the documentation). Only for testing, the participants gave a slight better ranking, see Fig. 17b. They mentioned in the free text field that knowing the individual use cases of an application is very helpful when writing end-to-end tests.

Finally, we asked the subjects whether they needed the visualization of all traces as in Fig. 15a and if so, for what purpose. The majority of 81.8% regarded a view of all dependencies as unnecessary. The subjects confirmed that the reduced number of dependencies as well as our two filtering options help in typical software engineering tasks. The remaining 18.2% gave reasons why they need such a visualization in corner cases of a software analysis. In the free text field of the questionnaire, they mentioned that this would help them to spot unused components or anomalous dependencies, for example ones that do not fit into any cluster.

Related Work

We visualize the static and dynamic aspects of a given software, regardless of its (original) design specification or its evolution history. Here, we sketch the differences between our work and related approaches. We intentionally do not discuss visualizations that focus on relationships between software versions and/or between an expected architecture and the actual code.

Static Aspects

Software City Layouts: The city metaphor maps software artifacts to city artifacts: components (e.g., classes) are buildings and containers of components (e.g., packages) are districts on which these buildings are located.

The TreeMap Layout [4–6, 19–21] is the most common layout for Software Cities. It uses binpacking to place rectangles (i.e., buildings and districts) into the smallest possible common rectangle and sorts them in descending order of their width, depth, or base area. The TreeMap layout considers only the hierarchical code structure.

Another way to layout a Software City is the Street Layout [19, 22] that is mainly used to visualize the development history. The buildings are organized by streets, classes of the same package are placed on the same street. However, Street Layouts also only consider the hierarchical code structure and require extensions that also show other dependencies as cluttered arcs atop the city. These layouts also suffer from the visual clutter and the overwhelming number of displayed arcs that we avoid.

Structure101: Structure101 [7, 8] is a tool to analyze software architectures. Its so-called Levelized Structure Maps (LSM) display dependencies among components in 2D. Similar to our approach, LSM also organizes the components into a stack of so-called levels. A component is shown in a level iff it depends on at least one component in the level directly below it. Components on the same level have no dependencies among them. Components transitively depend on others from the highest to the lowest level. Components on the lowest level do not have any dependencies. In the LSM representation there is also no need to use arrows to show dependencies, except for cyclic ones.

Graph Layout Techniques: Layered Graph Drawing organizes nodes in layers with most edges going in the same direction and with as few crossings and as few edges in the opposite direction. This is the key idea of our layout as well. Most layered graph drawing algorithms are based on the work of Sugiyama et al. [10] or its improvements [23, 24]. To avoid the NP-hardness of many of its steps (e.g., the Minimum Feedback Arc Set problem [11]), heuristics are used in practice. To avoid suffering from the complexity of this general problem, our heuristics exploit three properties of dependency graphs of software systems: (a) there are only a few cycles, (b) they are short, and (c) edges in general only belong to at most one cycle.

As far as we know, we are the first to apply layered graph drawing to layout Software Cities. We present domain specific heuristics that result in few feedback arcs, i.e., architecture violating dependency edges that cause cyclic dependencies in software.

Another common approach to make a graph easier to understand is Edge Bundling to reduce visual clutter [25–28]. But as we visualize most of the dependencies implicitly, there are too few arcs for such techniques anyway.

Dynamic Aspects

Trace Clustering: Thaler et al. [29] give a detailed overview and classify trace clustering techniques. We experimented with some common methods and determined that a density-based method is well suited for the trace data, but other methods may also fit. For visualizing dynamic software aspects it is not the clustering method that matters, but what it accomplishes. The clustering reduces the number of traces that we need to show explicitly as arcs, resulting in a less cluttered visualization. This allows us to visualize the dynamic behavior in the Layered Software City without losing clarity and comprehensibility.

Trace Visualizations: Common trace visualizations are graph representations [30–32], circular bundles [33], massive sequences [34], and hierarchical edge bundles [35]. To visualize traces in a Software City some approaches use straight lines between buildings [36], even with varying thickness [37], while most often arcs are spanned over buildings [38]. To avoid overloading the visualization with many traces edge-bundling is used [39]. SArF Map [40] uses a trace clustering to layout the buildings in a Software City and then visualizes all traces as edge-bundled arcs. Our approach reduces the number of explicitly displayed arcs as we only show one representative of a cluster and also provide filtering options.

Conclusion

To understand the functioning of a software system, one needs to understand the static dependencies among individual components. Showing all these dependencies explicitly, for example using arrows, leads to a confusing representation that is difficult to grasp. Based on ideas from layered graph drawing and using the well-researched city metaphor, this article presented a new layout for visualizing software. By encoding most dependencies in the layering, the proposed layout avoids all but those arrows that potentially indicate architecture violations. While minimizing the number of such so-called feedback arcs is a NP-hard problem, we presented heuristics that work well for cyclic dependencies in real software systems. In a controlled experiment we challenged professional software engineers with comprehension and refactoring tasks. They performed better (43%) and faster (5.3%) with the layered layout compared to the default layout of a Software City.

We extended the Layered Software City with a night view that displays dynamic dependencies as arcs atop buildings. To reduce the number of arcs and to preserve clarity, we suggested a clustering of similar traces and display only one representative trace per cluster, that usually corresponds to a use case of the studied application. Two additional filtering options, on the components' dependencies and on a single use case, further reduce the number of arcs. In a short study with professional engineers we investigated the usefulness of these two filters and learnt that 81.8% of the subjects do not need the visualization of all dependencies to perform software engineering tasks.

While our current work has been focused on software architects analyzing applications on their own, in the future we want to explore collaboration possibilities in the Software City to gain a better understanding in analyzing the software together. Open questions here include how to share information appropriately or how distinctive collaborators should be able to move around.

The source code of our visualizations and the raw data of the quantitative evaluations are available from https://github.com/qaware/holoware-software-city.

Funding Open Access funding enabled and organized by Projekt DEAL. No funding was received to assist with the preparation of this manuscript.

Declarations

Conflict of interest Veronika Dashuber receives a salary from company QAware GmbH. Michael Philippsen declares he has no financial interest.

References

1. Telea A. Data visualization: principles and practice. Boca Raton: CRC Press; 2008. https://doi.org/10.1201/b10679.
2. Weninger M, Makor L, Mössenböck H. Memory cities: visualizing heap memory evolution using the software city metaphor. In: Proc. 8th IEEE Working Conf. on Softw. Vis. 2020; pp. 110–121. IEEE.
3. Caserta P, Zendra O. Visualization of the static aspects of software: a survey. IEEE Trans Vis Comput Graph. 2011;17(7):913–33.
4. Alam S, Dugerdil P. EvoSpaces visualization tool: exploring software architecture in 3D. In: Proc. 14th Working Conf. on Reverse Eng., Vancouver, Canada. 2007: pp. 269–270.
5. Dhambri K, Sahraoui H, Poulin P. Visual detection of design anomalies. In: Proc. 12th Europ. Conf. on Softw. Maintenance Reeng., Athens, Greece, 2008: pp. 279–283.
6. Wettel R, Lanza M. Visualizing software systems as cities. In: Proc. 4th IEEE Intl. Workshop on Vis. Softw. Understanding Anal., Banff, Canada, 2007; pp. 92–99.
7. Headway Software Technologies Ltd: Levelized Structure Map (LSM), 2019. https://structure101.com/help/java/studio/Content/restructure101/lsm.html. Accessed: 10 Jun 2020.
8. Muccini H, Tekinerdogan B. Software architecture tool demonstrations. In: Proc. Working IEEE Conf. on Softw. Arch., Helsinki, Finland, 2012; pp. 84–85.
9. Dashuber V, Philippsen M, Weigend J. A layered software city for dependency visualization. In: Proceedings of the 16th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications, Online, 2021; pp. 15–26.
10. Sugiyama K, Tagawa S, Toda M. Methods for visual understanding of hierarchical system structures. IEEE Trans Syst Man Cybern. 1981;11(2):109–25.
11. Karp RM. Reducibility among combinatorial problems. In: Complexity of computer computations. New York: Springer; 1972. p. 85–103.
12. Zimmermann T. Changes and bugs—mining and predicting development activities. In: Proc. IEEE Intl. Conf. on Softw. Maintenance, Edmonton, Canada, 2009; pp. 443–446.
13. Hart SG, Staveland LE. Development of NASA-TLX (task load index): results of empirical and theoretical research. In: Human mental workload. Amsterdam: Elsevier; 1988. p. 139–83.
14. Hart SG. NASA-task load index (NASA-TLX); 20 years later. In: Proc. Annu. Meeting of Human Factors and Ergonom. Soc., Santa Monica, CA, 2006; pp. 904–908
15. Rousseeuw PJ. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. J Comput Appl Math. 1987;20:53–65.
16. Aranganayagi S, Thangavel K. Clustering categorical data using silhouette coefficient as a relocating measure. In: Intl. Conf. on Comput. Intelligence and Multimedia Applications. 2007; pp. 13–17. IEEE, Tamil Nadu, India.
17. Layton R, Watters P, Dazeley R. Evaluating authorship distance methods using the positive Silhouette coefficient. Nat Lang Eng. 2013;19(4):517–35.
18. Zhou HB, Gao JT. Automatic method for determining cluster number based on silhouette coefficient. Adv Mater Res. 2014;951:227–30.
19. Caserta P, Zendra O, Bodenes D. 3D hierarchical edge bundles to visualize relations in a software city metaphor. In: Proc. IEEE Intl. Workshop on Vis. Softw. for Understanding and Anal., Williamsburg, VA, 2011; pp. 1–8. https://doi.org/10.1109/VISSOF.2011.6069451.
20. Fittkau F, Waller J, Wulf C, Hasselbring W. Live trace visualization for comprehending large software landscapes: The ExplorViz approach. In: Proc. IEEE Working Conf. on Softw. Vis., Eindhoven, The Netherlands, 2013; pp. 1–4.
21. Vincur J, Navrat P, Polasek I. VR city: software analysis in virtual reality environment. In: Proc. IEEE Intl. Conf. on Softw. Quality, Reliability and Security Companion, Prague, Czech Republic, 2017; pp. 509–516.
22. Steinbrückner F, Lewerentz C. Representing development history in software cities. In: Proc. 5th Intl. Symp. on Softw. Vis., Salt Lake City, UT, 2010; pp. 193–202.
23. Dujmović Vea. On the parameterized complexity of layered graph drawing. In: Proc. Europ. Symp. on Algorithms, Århus, Denmark, 2001; pp. 488–499.
24. Eiglsperger M, Siebenhaller M, Kaufmann M. An efficient implementation of Sugiyama's algorithm for layered graph drawing. In: Proc. Intl. Symp. on Graph Drawing, New York, NY, 2004; pp. 155–166 .
25. Gansner ER, Hu Y, North S, Scheidegger C. Multilevel agglomerative edge bundling for visualizing large graphs. In: Proc. IEEE Pacific Vis. Symp., Hong Kong, China 2011; pp. 187–194.
26. Holten D, Van Wijk JJ. Force-directed edge bundling for graph visualization. Comput Graph Forum. 2009;28(3):983–90.
27. Pupyrev S, Nachmanson L, Kaufmann M. Improving layered graph layouts with edge bundling. In: Proc. Intl. Symp. on Graph Drawing, Konstanz, Germany, 2010; pp. 329–340.
28. Zhou H, Xu P, Yuan X, Qu H. Edge bundling in information visualization. Tsinghua Sci Technol. 2013;18(2):145–56.
29. Thaler T, Ternis SF, Fettke P, Loos P. A comparative analysis of process instance cluster techniques. In: Proc. 12th Intl. Conf. on Wirtschaftsinformatik, Osnabrück, Germany. 2015.
30. Knupfer A, Brunst H, Nagel WE. High performance event trace visualization. In: Proc. 13th Euromicro Conf. on Parallel, Distributed and Network-Based Processing, Lugano, Switzerland, 2005; pp. 258–263 .
31. Maoz S, Kleinbort A, Harel D. Towards trace visualization and exploration for reactive systems. In: Proc. IEEE Symp. on Visual Languages and Human-Centric Computing, Coeur d'Alene, ID, 2007; pp. 153–156.
32. Trümper J, Bohnet J, Döllner J. Understanding complex multi-threaded software systems by using trace visualization. In: Proc. 5th Intl. Symp. on Softw. Vis., Salt Lake City, UT, 2010; pp. 133–142. https://doi.org/10.1145/1879211.1879232.
33. Cornelissen B, Holten D, Zaidman A, Moonen L, van Wijk JJ, van Deursen A. Understanding execution traces using massive sequence and circular bundle views. In: Proc. 15th IEEE Intl. Conf. on Program Comprehension, Banff, Canada, 2007; pp. 49–58.
34. Elzen SVD, Holten D, Blaas J, van Wijk JJ. Dynamic network visualization with extended massive sequence views. IEEE Trans Vis Comput Graph. 2014;20(8):1087–99.
35. Holten D, Cornelissen B, van Wijk JJ. Trace visualization using hierarchical edge bundles and massive sequence views. In: 4th

IEEE Intl. Workshop Vis. Softw. for Understanding and Anal., Banff, Canada, 2007; pp. 47–54. https://doi.org/10.1109/VISSOF.2007.4290699.

36. Waller J, Wulf C, Fittkau F, Dohring P, Hasselbring W. Synchrovis: 3D visualization of monitoring traces in the city metaphor for analyzing concurrency. In: Proc. 1st IEEE Work. Conf. on Softw. Vis., Eindhoven, The Netherlands, 2013; pp. 1–4.
37. Fittkau F, Waller J, Wulf C, Hasselbring W. Live trace visualization for comprehending large software landscapes: The ExplorViz approach. In: 1st IEEE Work. Conf. on Softw. Vis., Eindhoven, The Netherlands, 2013; pp. 1–4. https://doi.org/10.1109/VISSOFT.2013.6650536.
38. Dugerdil P, Alam S. Execution trace visualization in a 3D space. In: Proc. 5th Intl. Conf. on Information Technology, Las Vegas, NV, 2008; pp. 38–43.
39. Caserta P, Zendra O, Bodenes D. 3D hierarchical edge bundles to visualize relations in a software city metaphor. In: Proc. 6th Intl. Workshop on Vis. Softw. for Understanding and Anal., Williamsburg, VA, 2011; pp. 1–8.
40. Kobayashi K, Kamimura M, Yano K, Kato K, Matsuo A. SArF map: visualizing software architecture from feature and layer viewpoints. In: Proc. 21st IEEE Intl. Conf. on Program Comprehension, San Francisco, CA, USA, 2013; pp. 43–52.