Visualization and Evolution of Software Architectures

Taimur Khan\textsuperscript{1}, Henning Barthel\textsuperscript{2}, Achim Ebert\textsuperscript{3}, and Peter Liggesmeyer\textsuperscript{4}

\textsuperscript{1,3} University of Kaiserslautern  
67653 Kaiserslautern, Germany  
\{tkhan,ebert\}@informatik.uni-kl.de  
\textsuperscript{2,4} Fraunhofer IESE  
67663 Kaiserslautern, Germany  
\{Henning.Barthel,Peter.Liggesmeyer\}@informatik.uni-kl.de

Abstract

Software systems are an integral component of our everyday life as we find them in tools and embedded in equipment all around us. In order to ensure smooth, predictable, and accurate operation of these systems, it is crucial to produce and maintain systems that are highly reliable. A well-designed and well-maintained architecture goes a long way in achieving this goal. However, due to the intangible and often complex nature of software architecture, this task can be quite complicated. The field of software architecture visualization aims to ease this task by providing tools and techniques to examine the hierarchy, relationship, evolution, and quality of architecture components. In this paper, we present a discourse on the state of the art of software architecture visualization techniques. Further, we highlight the importance of developing solutions tailored to meet the needs and requirements of the stakeholders involved in the analysis process.

1 Motivation

The field of software visualization is centered on visual representations aimed at making the software more comprehensible. These representations are a necessity for analysts to examine software systems due to their "complex, abstract, and difficult to observe" nature [53]. These difficulties are further compounded in large-scale software systems where it becomes increasingly difficult for analysts to examine its behavior and properties, due to the systems’ scale.

Software visualization focuses on various aspects of software systems, such as source code, software structure, runtime behavior, component interaction, or software evolution, to unravel patterns and behaviors through the different software development stages [1]. Due to the diverse nature of these data sets, different types of visualizations can be found in literature. However, for the focus of this research we highlight the visualization of software architecture as well as software architecture evolution.

The visualization of software architecture is an essential component of software visualization. "Not only are architects interested in this visualization but also developers, testers, project managers, and even customers" [32]. From the perspective of a software analyst, software architecture focuses on the structure of a software system - the focal point of which is to examine composing entities, their metrics, and relationships [11]. Additionally, recent studies have shown an increased interest in not only the visual exploration of software modules, their structure, and interrelations, but also in the evolution of these modules [19]. The key feature of software architecture visualization is to uncover visual metaphors that are both efficient and effective in depicting the software architecture of a system and to encode software code metrics within these representations. Several questions need to be addressed in finding such solutions, such as: who is the end-user of the architecture visualization [50], what needs to be analyzed through the visualization [52], and how can appropriate visualization metaphors and interaction techniques be chosen [2].
2 Visualization and Evolution of Software Architectures

One of the core topics in the field of software visualization is a means to effectively visualize, navigate, and explore the software architecture of a system [31, 32, 34]. Generally, object-oriented software tends to be structured hierarchically - with packages containing sub-packages, which in turn contain classes that hold methods and attributes. It is this hierarchy and relationships between software components that is of interest when it comes to software architecture visualization [15].

In this section, we explore representations of the global architecture of a system, such as tree, graph, and diagram model depictions. Further, we also investigate representations that highlight relationships between components as well as the importance of visualizing software metrics.

2.1 Architecture Representations

Tree structures are an ideal way of representing the hierarchical structure of software architecture. However, research in this area has shown the need to move forward from well-known techniques such as node-link layouts to more sophisticated ones to handle the larger hierarchies found in software systems nowadays [70]. Fig. 1 shows both a generic node-link diagram as well as one found in a commercial tool. Inspection of these representations shows that they quickly become too large and utilize available screen space far too poorly for proper investigation. Further, the amount of textual information represented in the nodes as well as the way relationships are depicted should be revisited to avoid visual clutter and information overload [41].

This section inspects several 2D visual representations [10] that may not be specific to just software visualization, but have been effectively applied to highlight the hierarchical structure of a software system [70, 4]. Here, it is important to note that a lot of these representations have been extended to 3D visualizations [2, 6, 49]. While 3D approaches have been shown to display larger hierarchies and minimize clutter [58], they have also suffered from the well documented drawbacks of 3D visualizations, such as: object occlusion, cumbersome view adjustments, performance issues, as well as poor readability of 3D texts [48, 17]. Due to these drawbacks and the requirements of our stakeholders, this survey focuses mostly on 2D representations.

The Treemap visualization (Fig. 2a), first introduced by Johnson and Schneiderman [39], is an effective means to visualize an entire software hierarchy. It is essentially a space-filling technique that displays hierarchical data as a set of nested rectangles. This is usually performed by a tiling algorithm that slices a box into smaller boxes for each level of the hierarchy, recursively, alternating between horizontal and vertical slices. “The resulting visualization displays all the elements of the hierarchy, while the paths to these elements are implicitly encoded by the Treemap nesting” [15]. In the context of software architecture visualization, Treemaps are used to represent methods as elementary boxes and classes as composed boxes. Several modifications of Treemaps appear in literature and in practise - some improve readability by enforcing an aspect ratio as close as possible to 1, while others have used irregular shapes such as Voronoi instead of rectangles to show more information [8]. Typically, designers are limited to the encoding of a single metric - the box color. While this provides a symbolic idea of how such a metric value
is spread through the hierarchy, it is not simple to determine or represent metrics of enclosing entities [22]. Treemaps provide an extremely compact layout, however, they are limited by mainly showing the leaves of the software structure. Similarly, the circular TreeMap visualization (Fig. 2b) and variations of it have been researched in order to have circles fill the available space [74]. However, as shown in Fig. 2b circular treemaps are not efficient with respect to the used space.

The Icicle Plot principle of Fig. 3a is where a line represents a tree level and each line is split according to its number of children [10]. While Icicle Plots provide better understanding of structural relationships as packages can be used as root and classes and methods as tree elements, scalability and navigation may be an issue with hierarchies of large systems [22]. Typically, two metrics maybe encoded in the visual representations: node size and color.

An alternative space-filling technique to nested geometry is the use of a Sunburst visualization that focuses on adjacencies instead [62]. This technique was first proposed by Stasko and Zhang [65], where they utilized a circular or radial display to depict the hierarchy rather than a rectangular layout (Fig. 3b). In a sunburst, the hierarchy is laid out radially with the root at the center and discs or portions of discs as deeper levels further away from this center [3]. In contrast to the Treemap techniques mentioned earlier and similar to the Icicle Plot, designers have the added flexibility to encode two distinct metrics: the angle swept out by an item and its color [22]. Studies have shown the performance of localization, comparison, and identification tasks in Treemap and Sunburst visualizations to be comparable, however the Sunburst is found to be easier to learn and more pleasant [64]. While screen-space is better utilized as compared to node-link diagrams, scalability and navigation may still be an issue in larger systems.

Another approach is to make use of the hyperbolic space, which intrinsically provides more space than a layout that employs Euclidean coordinates. This well-established technique is more commonly referred to as the hyperbolic tree layout (Fig. 3c) and was first introduced in the context of information visualization by Lamping et al. [43]. Essentially, it lays out the hierarchy in a uniform manner on a hyperbolic plane and maps the results back on to the Euclidean space. The resulting hierarchy is laid out on a circular display region and maybe complemented with focus and context techniques such as fisheye distortion [40], where components tend to diminish in size as they move outwards. This leads
to a larger representation of the center or focused area while still displaying the overall structure of the tree. Hyperbolic trees show detail and context at once; initially the root of the hierarchy is placed in the center, however, the display can be transformed to bring another node into focus through interaction. It would probably be best to encode metrics through the use of color alone, as varying the node size would adversely affect the layout algorithm. When the graph is deemed too large to be rendered effectively, nodes are pruned together and may be interactively expanded to reveal the subtree structure.

### 2.2 Visualizing Relationships

In contrast to visualizing the software hierarchy of a system, visualizing relationships of the software system is a more complex task. This is due to both the higher amount and the different types of relations that exist in a system, such as: inheritance, method calls, dynamic invocation, accesses, etc.

Generally, graphs have all the characteristics required to represent relationships of a software system. This is typically done by expressing software components as nodes and relationships between them as edges [63]. However, this often leads to the visualization of an extremely large graph due to the high interconnectivity between the large amount of components found in software systems nowadays. Thus, the resulting visualization tends to be extremely confusing and cluttered - it becomes difficult to discern between nodes and edges due to the cluttering, overlapping, and occlusion of edges (Fig. 4).

A well-known approach to remedy this clutter issue is to replace node-link diagrams with a square matrix that has matching row and column labels. The matrix then highlights the number of relations between row and column elements within each matrix entry, possibly through some visual representation [78]. This well-known technique is often referred to as the Dependency Structure Matrix [59] in literature and provides a compact and uncomplicated representation of relations in a complex system. However, keeping a mental map of the system hierarchy can still be an issue in these visualizations.

The most accepted graph-based software visualization in the field of object-oriented software engineering are UML class diagrams. This modeling language was created and developed by the Objected Management Group and has since become the industry standard for modeling software systems [28]. Its main purpose is to portray inter-class relations, such as: composition, inheritance, generalizations, aggregations, and associations. However, due to the amount of textual information depicted by each component such as the listing of methods and variables, these graphs grow exponentially with each additional component or class notation and are highly prone to information overload. Some researchers have looked at reducing the visual complexity associated with such graphs by reducing the number of overlapping edges, the use of orthogonal layouts, the horizontal writing of the labels, and edge bundling [24, 56, 68]. While some success in reducing the complexity has been achieved, the drawbacks associated with node-link diagrams such as poor screen-space management and information overload still need to be tackled.

Some researchers have experimented with different layout and filter techniques in order to resolve the clutter issue. An example of this is the work of Pinzger et al. [55] that focuses on the creation of condensed and aesthetically pleasing graphs that show information relevant to solve a given program comprehension task. Their solution was to use nested graphs and a feature that allowed to add and filter appropriate nodes and edges. Other researchers such as Holten [35] have chosen to implement better space-filling techniques in combination with improved edge representations. Holten’s approach was to
place software elements on concentric circles according to their depth in the hierarchical tree and then to
display edges above the hierarchical visualization (Fig. 5). Further, he extended the work of Fekete et
al. [26] that used spline edges to replace explicit arrow directions, in order to reduce the visual clutter
and edge congestion by allowing edges to bundle together according to a parameter (Fig. 5a and 5b).
Similarly, techniques displaying, clustering, and filtering edges on top of structural representations can be
utilized in other visualizations (Treemaps, circular trees, etc) to represent the hierarchical graph structure
of a software system.

Another approach to resolve the issues of cluttered 2D graphs is the use of 3D visualizations [29],
where the user can access a view without occlusions. However, 3D representations of large graphs have
their own problems, such as: navigation can not only be difficult but also disorienting [60], object occlu-
sion, performance issues, and text illegibility [48]. For the purpose of completion it would be prudent to
mention some of the more prominent work in the area of 3D software architecture visualization. Some
researchers in this field have experimented with real-world metaphors to take advantage of the intuitiv-
ness of these representations [51]. For example, the City or Cities metaphors are often used to depict
relationships through a visually understandable metaphor [2, 52], where cities (packages) are connected
via streets (two-directional calls) and water (uni-directional calls). Similarly, researchers have realized
the Solar System [33], Island [52], and Landscape [6, 9] metaphors, where the respective relationships
between each contributing element is exploited to depict packages, classes, and their relationships.
Another interesting approach towards handling large and complex graphs is the clustered graph layout (Fig.
6), where clustering, dynamic transparency, and edge bundling are used to visualize a graph without
altering its structure or layout [7].
2.3 Visualizing Software Metrics

The incorporation of software metrics is an important component in the analysis of a software systems architecture, as they not only provide an insight into the quality of the software design [14, 27] but also a means to monitor this quality throughout the design process [12]. Typical static software metrics express different aspects of a complex system, such as: design complexity, resource usage, and system stability.

The idea behind metric-centered visualizations is to transform numerical statistical data into a visual representation that is easier to understand and grasped far more intuitively and instantaneously [75]. Here, the greatest challenge is to find an effective mapping from a numerical representation to a graphical one that enhances the structural visualization [38].

In this section, selected visualization techniques that implement static software metrics are highlighted - the purpose of which is to provide an idea of the implemented approaches. One such approach is to combine them with UML class diagrams. An example of this is the **MetricView** (Fig. 7a) visualization that displays metric icons on top of UML diagram elements [71].

An extension of this approach is the **areas of interest** (Fig. 7b) technique developed by Byelas and Telea [13]. They apply a layout algorithm that groups software entities with common properties, encloses these entities with a contour, and adds colors to depict software metrics. In order to distinguish overlapping areas, each area is given its own texture, such as: horizontal lines, vertical lines, diagonal lines, and circles. Further, shading and transparency techniques are used to improve the distinction between several areas.

In visual representations other than UML Diagrams, similar approaches have to be implemented in order to combine metrics and structural information. An example of this is the work of Holten et al., where they used texture and color to show two different software metrics on a Treemap [37]. Their results show that the combination of color and texture provides high information density, assists in finding correlations between metrics, and can reveal patterns and potential problem areas.

To visualize multiple aspects of a software system, Lanza et al. introduced the concept of **polymetric views**, where the visualization of a software is enriched with software metrics [46]. Essentially, they propose a node representation that encodes up to five distinct metrics: node width, height, x and y-coordinates and color, and edge width and color. They applied this to an inheritance tree where nodes represent classes and edges depict the inheritance relationship between them. Node width and height is used to encode the number of attributes and the number of methods. Further, a color tone is applied to represent the number of lines of code.

Similarly, in 3D visualizations the encompassing visual entities have been encoded with software metrics [33, 76]. Another technique that may be applied in the analysis of system metrics is use of filters, an example of this can be found in the Solar system metaphor, where filters may be applied to the overall system to visualize planets with metric values that lie within a chosen interval [33].
3 Visualization of Architecture Evolution

A general obstacle with regards to software evolution visualization is coping with the complexity that emerges from the huge quantity of evolution data; it is quite common to have hundreds of versions of thousands of files [72]. The technical challenges associated with extrapolating this historical data are deemed out-of-context with respect to this paper, instead, the focus will be on visualizing the evolution of the software architecture.

Real software solutions undergo continuous change to meet new requirements, adapt to new technology, and to repair errors [47]. Inevitably, the software in question magnifies in both size and complexity, often leading to a situation where the original design gradually decays unless proper maintenance is performed [20]. As such, visualizing the evolution of the software architecture is one of the key topics in the field of software evolution visualization [15]. It is essential to have a global overview of the entire system evolution in order to explain and document how a system has evolved to its present state and to predict its future development [18].

This section follows the same pattern as the previous one, where we first focus on how the global architecture of the software changes with each release and then examine how relationships and metrics evolve within each version.

3.1 Visualizing Hierarchical Changes

Since software maintenance is performed mainly at code level, most visualizations have implemented a 2D line-based approach to represent the software evolution [25, 69, 73]. Generally, the adopted approach is to visually map a code line to pixel line, where color is typically used to show the age of a code fragment [25]. Additional focus has been to develop interaction techniques that allow users to effectively navigate and explore the data [73]. In order to highlight the state of the art in this traditional approach, the Code flows visualization technique [69] is briefly examined. Fig. 8a shows an evolution from left to right of four versions of a source code class. This technique employs an icicle layout and bundled edges to show how a source code line changes over subsequent versions. Source code lines that do not change from one version to another are colored black, while code lines that changed are highlighted using different colors. In general, these tools are successful in tracking the linebased structure of software systems and reveal change dependencies at given moments in time [73]. However, they lack the sophistication to provide insight into attribute changes and more so the structural changes made throughout the development process.

In contrast, there are only a few visualizations aimed at representing structural changes of a system architecture over time [15]. As explained earlier, there definitely exists a requirement to monitor the evolution of a systems architecture, however, current graph animation algorithms are limited and need to...
One such approach, is the work of Holten et al. that presents a technique aimed at comparing the software hierarchies of two software versions [36]. To better compare the two versions, the algorithm tries to position matching nodes opposite to each other. This technique is presented in Fig. 8b, where the source code of Azureus v2.2 is displayed on the left and v2.3 is portrayed on the right. Nodes that are present in one version but not the other are highlighted via red shading. Further, the Edge Bundles technique of Section 2.2 is used to highlight and track the selected hierarchy.

Collberg et al. describe a system that visualizes the evolution of a software system using a graph drawing technique that handles a temporal component for the visualization of large graphs [16]. They accomplish this by utilizing a force-directed layout to plot call graphs, control-flow graphs, and inheritance graphs of Java programs. Changes that the graphs have gone through since inception are highlighted through the use of color. Nodes and Edges are initially given the color assigned to its author (red, yellow, or green) and progressively age to blue (Fig. 9a).

Lately, there has been some effort by researchers to extend known metaphors to handle the evolution of software systems. Steinbrückner et al. have an interesting approach that implements the city metaphor for the representation of large software systems in the form of evolving software cities [66]. Their work is illustrated in Fig. 9b, where a system grows from an initial 389 classes to 439 classes in revision 100 and 466 classes in revision 200. In this implementation of the city metaphor, streets represent Java packages and building plots represent Java classes. The sequence of visual depictions aims to highlight basic changes in the software structure, how elements maybe added, removed, and moved within the software hierarchy. Further, they extend this general representation to address the needs of two distinct application scenarios by: 1) applying an evolution map that uses contour lines to show different versions of each subsystem and 2) using a modification history map that uses a contour line map combined with property towers that depicts the number of modifications as height and modification date as color.
3.2 Visualizing Software Metrics Evolution

As covered in Section 2.2, visualizing relationships is an extremely complex task that is further compounded in the case of software evolution. Typically, researchers and practitioners focus more on the logical coupling between source code artifacts, as it can be encoded easily into metric values [30].

Software metrics are an ideal abstraction as they encapsulate, summarize, and provide essential quality information about source code [44]. As such, they are essential in providing a continual understanding and analysis of the quality of a system during all phases of the product life cycle. Instead of tedious, inefficient, and hard to grasp numerical representations, metrics tend to be mapped to graphical characteristics so that they may be intuitively interpreted. In this section, we explore the state of the art in the visualization of software metrics across different software versions.

The Evolution Matrix is a visualization technique that provides an exploratory view of an object-oriented systems evolution, both at the system and class granularity levels [45]. In this work, Lanza et al. combine software visualization and software metrics by using two-dimensional boxes to represent classes and encoding metric measurement of the classes to the width and height of the boxes. In the example of Fig. 10a, they use the metric number of methods for the width and number of instance variables for the height, columns to represent different versions of the software, and rows to depict different versions of the same class. At the system level, this technique recovered the following characteristics regarding the evolution of a system: size of the system, addition and removal of classes, and growth and stagnation phases in the evolution. While at the class level, it shows if the class grows, shrinks, or stays the same from one version to another. These features allow the expert to analyze a number of interesting aspects, such as a class growing and shrinking repeatedly, a class suddenly exploding in size, or a class that had a certain size but lost its functionality.

The visualization framework by Langelier et al. also facilitates the analysis of software over many
versions [44], albeit in a slightly different manner. Instead of employing a technique that displays the entire system evolution in one picture [45], they rely on animated transitions from one version to another. As Fig. 10b shows, there are different static representations for each subsequent version; the image on the left is a previous version and the image on right is the next. The user controls forward and backward navigation in time, which in turn animates three graphical characteristics that are mapped to metric values - color, height, and twist. While the animations are of a short duration, they are well-designed and help attract the attention of the viewer towards program modifications [44]. This work of Langelier et al. contains references to extensive case studies aimed at detecting both evolution patterns and known anomalies. With respect to evolution patterns, users were able to identify constantly growing classes, quick birth and death of classes, and explosions in complexity in a short time-span. On the other hand, while looking for common anomalies, patterns such as the **God Class** or **Shotgun Surgery** were observed. The former is detected when a class constantly grows in complexity and coupling, while the latter occurs when a class constantly grows in terms of coupling and whose complexity increases globally but with an up-and-down local pattern.

Wettel and Lanza present interactive 3D visualizations in their **CodeCity** tool that examines the structural evolution of large software systems at both a coarse-grained and a fine-grained level [77]. At a coarse-grained level of granularity, classes are shown as monolithic blocks that lack details of the internal structure. While at the fine-grained level, the focus is on methods that appear as building bricks. Fig. 11a shows this fine-grained representation, where classes are illustrated as buildings located in districts that represent the packages in which the classes are defined. Metric values are then encoded in the visual properties of the city artifacts; class properties such as the number of methods and number of attributes are mapped on to the buildings’ height and base size, package depth is mapped on the districts’ color saturation. Further, the age distribution of classes is represented through an Age Map color mapping, where the color scheme ranges from light-yellow for recent entities to dark blue for earlier versions. Similar to the work of Langelier et al., back and forth transitions through the history of the system allows the city to update itself and reflect the currently displayed version. Additionally, at a finer level-of-detail the entire evolution of a single class or package may be tracked (Fig. 11b).

Pinzger et al. introduced a multivariate visualization technique that can display the evolution of numerous software metrics related to modules and relationships [54]. This is accomplished through a combination of graphs and **Kiviat diagrams** to graphically represent several metric values by plotting each value on its corresponding line (Fig. 12). The individual Kiviat diagrams present quantitative metrics, where low values are placed near the center of the Kiviat diagram and high values are found further away from the center. Dependency relationships between source code entities is highlighted by the layout of the
Figure 12 Kiviat graph with 20 metrics, 7 modules, and 7 subsequent releases [54]

The main aim of these tools is to employ a combination of metaphors and techniques presented in this paper to assist technical users, project managers, and researchers in analyzing software architectures. The study of Telea et al. shows that the mainstream masses are starting to realize the potential of these visualization techniques. For example, tools such as Lattix and NDepend have incorporated newer diagram-layout techniques, realizing the limitations of traditional node-link diagrams [70]. However, this modernization of AVTs is much slower than the advent of cutting-edge visualization solutions.

AVTs typically support a combination of the following tasks: "comparing desired and actual architectures, identifying architecture violations, highlighting architecture patterns or layers extracted from code bases, assessing architecture quality, and discovering evolutionary patterns such as architectural erosion" [70]. However, no single tool can satisfy all these needs and requirements, as they differ in the features they provide, the audience they cater to, and the tasks they support [50, 70].

The reader may refer to the work of Babu et al. [5] for a thorough comparison of AVTs according to the taxonomies they support. A closer inspection of these taxonomies is required, as it is imperative that visualizations are constructed to address problems and issues faced by the users of the system, rather
than just provide 'pretty pictures'. The challenge often is that different stakeholders, such as: architects, developers, maintainers, and managers, require contrasting tools and techniques to delve into different levels of details. In the context of software architecture, several researchers, such as: McNair et al. [50] and Panas et al. [52], have conducted in-depth analysis of what to visualize and how best to achieve it. A good synopsis of these findings can be found in the survey of Ghanam et al. [32].

The most significant lesson learnt from the above-mentioned surveys is not to lose sight of the audience and to conduct appropriate evaluations where possible to determine the true worth of a proposed software architecture visualization; does it allow for a more thorough analysis (number of issues detected) or for a more efficient one (task completion time).

5 Conclusion

In this paper, we provide a comprehensive and up-to-date review of both literature and mainstream practises in the field of software architecture visualization. Our research shows that the architecture visualization domain has evolved significantly in recent years giving developers new tools to better understand, evaluate, and develop software and helping managers to monitor design and refactoring issues. However, there remains the need to incorporate these cutting-edge tools and techniques with standard software development and maintenance practices.

Some visualization techniques like parallel coordinates and bundled diagram layouts are less-known in industry, while other techniques such as node-link layouts are well-known. The software architecture community has not made widespread use of these recent advances. There is a definite need to bridge this gap, as software systems are getting far too large to be analyzed through traditional means alone. This delay in adopting new technology may be due to the stakeholders not having enough time to try out every new tool, lack of knowledge with respect to technical visualization terms often used in marketing these tools, or simply a reluctance to try unknown visualization metaphors and techniques.

The way forward is for researchers to work closely with experts, tailor tools to meet specific requirements, and to conduct comprehensive evaluations. This would lead to research prototypes making their way into mainstream tools and a widespread adoption. It is envisioned that this transition would improve quality and reduce the time and cost factors. Lastly, we would like to point out the need for both industry and academia to look into the evolution of software at higher level of abstraction than current linebased methods; this remains an open area for future research.

References

Visualization and Evolution of Software Architectures


28 UML Forum. Uml faq @ONLINE, December 2011.


40 Alan Keahey. A brief tour of nonlinear magnification @ONLINE, November 2011.


Werner Randelshofer. Visualization of large tree structures @ONLINE, November 2011.


16 Visualization and Evolution of Software Architectures


