Texture-Based Metrics Visualization on Software Architecture Diagrams

Heorhiy Byelas* Alexandru Telea†

Institute of Mathematics and Computer Science
University of Groningen, the Netherlands

Abstract

In this paper, we address the problem of visualizing several types of metrics computed on software architecture diagrams. Our specific aim is to present metrics computed on groups of diagram elements, such as classes or components in UML diagrams, together with metrics computed on diagram element members, such as class methods. For member metrics, we use an adapted version of the known table lens technique. For group metrics, we design a new solution that uses blended textures computed via spatial interpolation to show metric variations. Our method helps the task of visually correlating the distribution and outlier values of a multivariate metric dataset with the diagram structure. We present applications on architecture diagrams extracted from real-world software systems.

1 Introduction

Architecture diagrams and software metrics are essential components in software engineering activities such as forward and reverse engineering. Diagrams capture structural and functional relations between different system elements, such as interfaces and implementations. Software metrics capture several types of aspects of the modeled system, from design-phase attributes (*e.g.* design and code quality) to maintenance attributes (*e.g.* testability, maintainability and evolvability) and run-time data (*e.g.* performance execution). In UML class diagrams [OMG 2008], metrics can be associated with classes, relations between classes, groups of classes, or class members, *e.g.* methods. In all such activities, a recurrent question is how to easily correlate several metrics computed on a system with the system structure, captured by its architecture diagram.

We address here this question by presenting a new way to visualize several software metrics, modeled as a multivariate dataset, atop of a software architecture diagram, in a single picture. We focus here on two types of metrics: *member metrics*, defined on diagram element members (*e.g.* class methods) and *area-of-interest metrics*, defined on groups of diagram elements, which are also called areas of interest (AOIs) [Byelas and Telea 2006]. Our key idea is as follows. We use blending, texturing, and smooth scattered point interpolation to render several AOI metrics, defined as a multivariate dataset with potentially missing values, atop of areas of interest in diagrams, so that users can spot metric-metric and metric-area correlations. This frees the space within the diagram elements, which can be employed to show member metrics, using a table lens technique [Rao and Card 1994].

This paper is structured as follows. Section 2 reviews related work in visualizing the combination of system structure and software metrics. Section 3 presents our new technique for rendering several metrics atop of AOIs. Section 4 presents two examples of using our new rendering technique on three UML diagrams, and also demonstrates the use of the table lens technique for method metrics. Section 5 discusses our results. Section 6 concludes the paper.

2 Related work

In practice, software system structure is visualized using various types of node-link diagrams. Within this large class, UML diagrams are arguably the most accepted metaphor for object-oriented and component-based systems [OMG 2008]. Metrics can be computed by static analysis tools [Wust 2006], simulation tools [Bondarev et al. 2006] or dynamic analysis tools such as profilers or debuggers. In the engineering practice, metrics are usually shown as separate numerical tables, making their correlation with large diagrams difficult and time-consuming.

Several attempts were made to combine metrics and UML-like diagrams. Lanza *et al.* render class-level metrics by mapping them to the class size and/or color [Lanza and Marinescu 2006]. In this way, two metrics can be shown simultaneously. Similar techniques are used by many software visualization tools, such as the well-known Rigi framework [Tilley et al. 1994]. An extensive overview of such tools is given by Diehl [Diehl 2007]. However, in typical UML diagrams, element sizes are constrained (fixed), so they cannot be used to show a metric. This happens when UML diagrams are created in the design process: designers carefully craft the layout of diagrams and would not accept the elements' sizes or positions to be modified, as this destroys their 'mental map'. Element background colors can also be constrained *e.g.* when we want to draw method names or other text annotations inside each class. Termeer *et al.* show UML class-level metrics with icons scaled, colored, and drawn atop classes [Termeer et al. 2005]. This technique can show individual metric values and helps spotting outliers. However, correlating *several* metrics on a *large* diagram is difficult, as one has to memorize the metric values while visually scanning the diagram.

Moreover, all the above-mentioned techniques address only metrics defined on diagram elements (*e.g.* classes) or members (*e.g.* methods). We also want to visualize metrics defined on *groups* of elements, also called areas of interest (AOIs), *e.g.* safety of all multithreaded components, speed of all performance-critical components, and so on. Byelas and Telea addressed the problem of showing the AOIs themselves by surrounding the grouped elements with a smooth contour, similarly to the way humans draw such groups on paper diagrams [Byelas and Telea 2006]. This shows which elements are in which AOI, *e.g.* all multithreaded elements in our example. However, the problem of showing metrics defined on AOIs, to correlate metrics with structure, is still open.

3 Drawing area-level metrics: Data model

To explain how we render metrics on AOIs, we first introduce our data model. Consider a diagram with n areas of interest $A_1 \ldots A_n$ defined over its elements, where $e_{ij}, j \in [1, |A_i|]$ are the elements in area A_i and $|A_i|$ is the number of elements in area i. For each A_i, we have a metric $m_i : [1, |A_i|] \rightarrow \mathbb{R} \cup None$ defined over its elements[1]. m_{ij}, the value of m_i on e_j, can have either a numerical value, or *None*, if that value is missing. Missing metric values are frequent in software analysis, *e.g* due to various limitations of the analysis tools [Wust 2006]. The set of metrics m_i can be seen as a multivariate scattered-point dataset [Spence 2007], with elements j as data points and the metric values i as variables.
We want to show all metric values for all areas in one image, so that

- we can compare the metric values of each element,
- we can visually follow how a metric varies over an area,
- we see the elements having missing values,

*e-mail: h.v.byelas@rug.nl
†e-mail: a.c.telea@rug.nl

[1] Several metrics defined on the same area are handled analogously

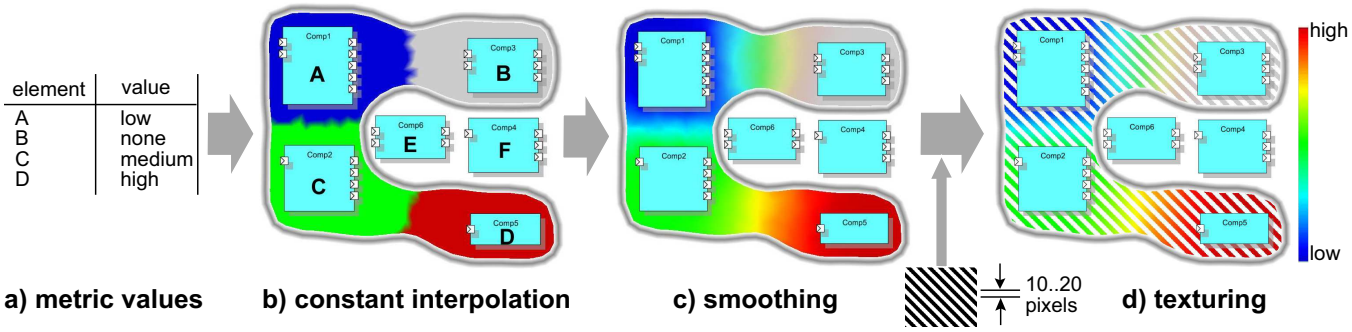


Figure 1: *Smooth interpolation of element metrics over an area-of-interest*

- we do not draw metrics on the elements themselves.
- we use, but do not change, a given diagram layout

We can visualize the AOI itself (but not its metrics) as shown by [Byelas and Telea 2006], by drawing a contour that encloses the elements located in the AOI (see *e.g.* Fig. 1 b). We show next how to render several metrics defined on several such AOIs so that metric values and areas can be easily correlated. We use a two step solution. First, we render the values of a single metric m_i over a given area A_i (Sec. 3.1). Next we combine all metrics m_i for all areas A_i in a single image (Sec. 3.2). Finally, we add shading to the areas to further emphasize their structure (Sec. 3.3).

3.1 Rendering a single metric

Termeer *et al.* show element metrics using icons scaled and colored by metric values, drawn atop of the elements [Termeer et al. 2005]. This has several drawbacks. Consider a diagram with five metrics over five areas of interest (Fig. 5). First, icon sizes are constrained by the element sizes (which can be very small), so it is hard to see the specific metric values. Second, we want to keep the element surfaces free to draw other data, such as method names and method metrics, as we shall see in Sec. 4.2. Third, correlating metrics with areas of interest, *e.g.* seeing how metric values change over one or several areas, is difficult, since there is no explicit visual correspondence (mapping) from the metric icons to areas.

We address these issues by rendering metric values *outside* the diagram elements. Denote by $\{e_i\}$ the elements in area A, with metric values m_i - we drop area-indexes here since we consider a single area. We encode missing metric values in a separate dataset $p_i : [1..|A|] \rightarrow \{0,1\}$, *i.e.* set p_i to 0 if m_i is missing, else set p_i to 1.

Our key idea is to produce an interpolation function $\mathscr{M}$ of the values m_i over area A. $\mathscr{M}(x)$ should equal the given metric values m_i for points x inside or close to the elements e_i, and vary smoothly in-between. We compute $\mathscr{M}$ as follows. First, we compute the Delaunay triangulation of A using the Triangle library [Shewchuk 1996]. Next, we initialize $\mathscr{M}$ at each triangulation vertex x with the metric value $m(e_{closest})$ of the element

$$e_{closest} = \underset{i \in [1..|A|], m_i \neq None}{\operatorname{argmin}} (||e_i - x||)$$

i.e. the closest element to point x which has a metric value. This yields an approximation of the Voronoi diagram of the element set $\{e_i\}$, so $\mathscr{M}$ is a piecewise-constant interpolation of $\{m_i\}$ over A. Figure 1 b shows $\mathscr{M}$ for the metric values in Fig. 1 a, using a red-to-blue colormap[2]. Element D has a maximum value, as shown by the surrounding red color. Element A has a minimum value, shown by the blue color. Elements E and F do not belong to the area. Element B, although inside the area, has no value. We show this using a neutral gray hue, as follows. We compute an interpolation $\mathscr{P}$ of the set $\{p_i\}$ over A, just as the interpolation $\mathscr{M}$ of $\{m_i\}$. With $\mathscr{M}$ and $\mathscr{P}$, we now compute the hue-saturation-value color of any point $x \in A$ as

$$h(x) = rainbow(\mathscr{M}(x)) \quad (1)$$
$$s(x) = \mathscr{P}(x) \quad (2)$$
$$v(x) = 1 \quad (3)$$

where *rainbow*() is the chosen colormap (see Fig. 1 d). Hence, points having metric values are rendered with saturated colors, while points with missing values are gray. Finally, we render the area's border using a soft gray texture.

In the final step, we smooth our piecewise-constant interpolation. For this, we apply a Laplacian filter [da Fontoura Costa and Cesar 2004] on $\mathscr{M}$ and $\mathscr{P}$, by setting the value of each triangle vertex x to the average value of all vertices connected to it, and repeating the process for 30..50 iterations. The points contained inside the elements e_i are kept fixed to the prescribed metric values m_i, to enforce the interpolation's boundary conditions. The result shows the values m_i close to their elements e_i, smooths values in-between, and grays out colors close to elements without values (see Fig. 1 c).

3.2 Combining several metrics

Now we must combine several metrics defined on possibly overlapping areas. We cannot simply additively blend areas of different colors as in [Byelas and Telea 2006], as this would mix the individual colors which show metric values beyond recognition. We use a texture-based solution: For each area A_i, we use a different texture[3]. We carefully designed a small set of textures (Fig. 2). The overlap of any textures in this set creates a visually different pattern. The textures contain just opacity: black denotes opaque zones, white gaps are fully transparent, gray indicates an alpha value between 0 and 1.

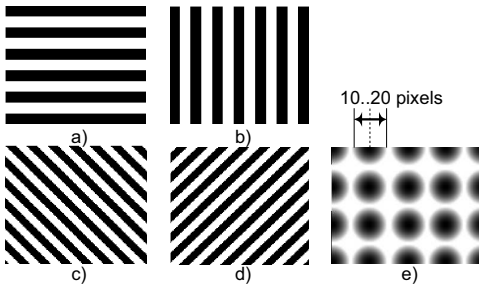


Figure 2: *The proposed set of textures. Gray value denotes opacity*

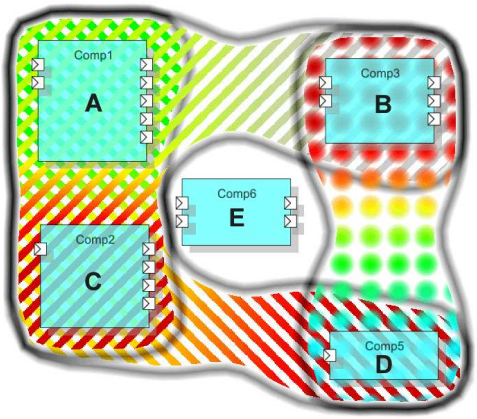
[2] The colormap choice is discussed further in Sec. 5

[3] We can relax this: non-overlapping areas can use the same texture, see Fig. 6.

We now render each area A_i by combining its color (showing metrics) computed by interpolation (Sec. 3.1) with its transparency texture (showing the area's identity) using OpenGL's texture modulation. Figure 1 d shows the application of texture c from Fig. 2 on the area in Fig. 1 c. To maximize information visibility, we draw areas starting from the largest to the smallest one, so that small areas appear atop, or possibly nested within, large areas.

Figure 3 shows three overlapping areas defined over four elements. Transparency creates hole-like patterns that let us see which textures, *i.e.* which areas, overlap, since each area has a different texture. The visual 'weaving' of the textures also lets us distinguish their different colors, hence correlate metric values. For example, we see that D has low values in area 1 and high values in area 3 - blue circles atop red diagonal lines; B has high values in area 1 and no value in area 3 - red circles atop gray diagonal lines; and so on.

Transparency acts more like a stencil, so there is little or no actual blending; colors do not mix, but get spatially woven. Color interpolation spreads the metrics information from elements over entire areas, creating large smooth hue spots which are easier to follow than rapid changes. We acknowledge this is a controversial issue: color blending may suggest that there is a continuous metric variation over an AOI, which is not the case. If less blending is perceived as better, one can simply do less smoothing iterations: See *e.g.* Fig. 10 where only a few iterations are done, which yields well-separated color areas around the elements, and almost no color interpolation. For instance, the transition between blue and green in A_2 is sharp and quick . Also, one can use discrete (categorical) colormaps with no change in our method, if these are believed to produce easier interpretable results.



Area	Elements	Texture
A_1	B, D	
A_2	A, B, C	
A_3	A, C, D	

Figure 3: *Diagram showing three areas of interest with metrics*

3.3 Shading for enhanced area separation

Although each area has its own distinctive texture, this can create confusing overlaps where it is hard to tell where an area exactly stops and another one starts. This happens *e.g.* where contours of different areas run almost tangent.

To alleviate this, we emphasize each area A by *shading*, as follows. We construct a signal $\mathscr{S}$ over A that is zero on the contour ∂A of A, one further from the contour, and varies smoothly with the distance within a narrow band of thickness δ along the contour. We compute $\mathscr{S}$ on the same triangle mesh as $\mathscr{M}$ and $\mathscr{P}$ used for the color interpolation (Sec. 3.1), as follows. First, we set $\mathscr{S}$ to 0 on the contour vertices and 1 elsewhere. Next, we use the same Laplacian filter as for color smoothing, keeping $\mathscr{S}$ fixed to 0 on the contour points, for 10..30 iterations. More iterations increase the thickness of the shading effect. After each iteration, we renormalize $\mathscr{S}$ to the range $[0,1]$.

We now use $\mathscr{S}$ as luminance by setting $v(x) = \mathscr{S}$ in Eqn. 3. This darkens areas close to their borders, but keeps them bright in the middle. Normalization ensures that shading is always bright in the middle of an area and dark on the contour. A direct application of shading would only affect the texture stripes (non-transparent) but would not show up in the texture 'holes'. This would create a broken, distracting shading effect. We prevent this by increasing the holes' opacity in the texture patterns from 0 (fully transparent) to 0.2 (slightly opaque).

Overall, we obtain the effect of convex, shaded 3D shapes - compare Fig. 3 (no shading) with Fig. 4 (with shading). At overlaps, the shaded shapes get woven by blending. The darkened borders help to visually separate areas (see the images in Sec. 4). The slight opacity of the texture pattern holes is able to show the shading close to the areas' contours and also a faint hue of the interpolated colors, *i.e.* metrics, in the pattern holes. This further strengthens the visual cohesion of all elements within an area and limits the breaking effect of the holes, but still allows pattern weaving to take place. When using textures to show metrics, as users noted on several occasions, textures seem to complicate the visual tracking of an area's contour, so shading has a stronger value in helping users to separate the areas.

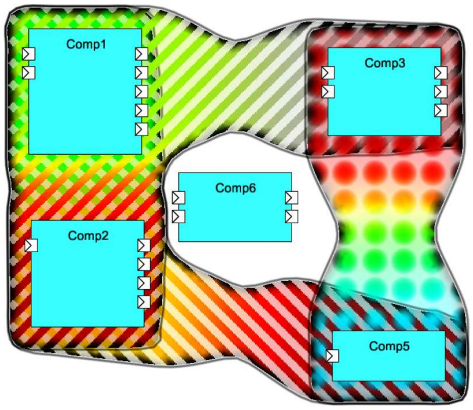


Figure 4: *Enhanced areas using shading (compare with Fig. 3)*

4 Applications

We now illustrate the use of our multivariate metric-and-structure visualization in two different case studies.

4.1 Case Study - JPEG Decoder Architecture

We consider a real-world software project: the architecture of a component-based JPEG decoder [Bondarev et al. 2007; Trust4All 2005]. The system model was built and its operation numerically simulated using the CARAT toolkit [Bondarev et al. 2006]. This delivered several run-time performance metrics. We next show two such metrics:

- μ_{CPU}: CPU usage for active components (each active component has its own process)
- μ_{mem}: memory usage for passive components (a passive component is used by active processes)

Given the actual architecture of the JPEG decoder, not all components have both memory and CPU metric values.

The decoder performs five tasks ($T1 \ldots T5$): JPEG stream starter ($T1$), inverse discrete cosine transform (IDCT), IDCT column process ($T2$), IDCT row process ($T3$), rasterization ($T4$), and rendering ($T5$). For a detailed description, we refer to [Bondarev et al. 2007]. We consider six areas: $A_1 \ldots A_5$ contain the components in tasks $T1 \ldots T5$. Each component has a memory usage metric μ_{mem} for each task area it is part of. The sixth area A_{CPU} holds all active components, which also have a CPU usage metric μ_{CPU}. We now address two goals which were named as important by the architects:

- understanding the distribution of tasks over the system structure and the memory usage of passive components
- understanding the CPU utilization over different tasks

To illustrate the advantage of our method, we first use the existing metric icons technique [Termeer et al. 2005]) to show the memory usage metric. First, we draw the areas $T1 \ldots T5$. Next, we draw pie and height-bar icons colored by task and scaled to show memory usage μ_{mem} (Fig. 5). The metric legend shows the tasks' colors and also shows where each icon from each task-area is placed within each element (see [Termeer et al. 2005]). The results are clearly not easy to interpret: In Fig. 5 it is hard to tell the metric values of each component for each area it belongs to. We cannot increase icon sizes, as each icon already takes one-sixth of a component's size. It is hard to visually correlate metric values over large areas. Also, a missing icon has an ambiguous meaning: does it show a zero $\mu_{mem} = 0$ or missing metric value or a missing metric value $\mu_{mem} = None$?

We now use our proposed technique. Each area (task) uses a different texture (see legend in Fig. 6 left). Color shows the memory usage μ_{mem} (blue=low, red=high). We now better see which value μ_{mem} each component has in each area, even though the images in Fig. 6 are half the size of the icon-based visualization in Fig. 5. We see, for instance, that components A, C, D, E and F use much more memory than the rest in at least one task they are involved in. Components A and C consume high memory amounts in the tasks they are involved in ($T1$ and $T3$ for A and $T2$ and $T4$ for C). Component C is the main memory consumer of the entire system, as both textures surrounding it are red. Indeed: C implements the decoder's pixel raster buffer, which consumes a lot of memory. Finally, we see that components $D \ldots F$ have a similar memory usage pattern: low in task $T4$, high in task $T5$. The results match the design expectations, as rendering ($T5$) is more memory-demanding than rasterization ($T4$).

In our second scenario, we add the CPU utilization metric μ_{CPU} (Fig. 6 right). The area A_{CPU}, containing all active components using CPU cycles ($G \ldots K$), intersects the task-areas $T1 \ldots T5$. To visually segregate the two aspects (tasks and CPU utilization), we use diagonal stripes for the task-areas $T1 \ldots T5$ and vertical stripes for the CPU utilization area A_{CPU}. We see now the CPU-intensive components: J and K. We also see that all components in A_{CPU} miss memory consumption data: the diagonal stripes textures around all components ($G \ldots K$) are gray (Fig. 6 left). This is correct, as the design of this JPEG decoder splits data (passive) components from algorithm (active) components.

Figure 7 shows the effect of adding shading. The left image depicts the six areas with color interpolation (showing metrics) but no textures. We provide this image to emphasize the useful effect of shading in understanding area overlaps. The right image shows the six areas and two metrics (memory and CPU usage). Compared to Fig. 6 right, it is easier to tell in the shaded image which components are in which areas.

4.2 Case Study - Large Class Diagrams

In our second application, we extracted an UML class diagram from the source code of a C++ graphics editor in a reverse engineering process, using an ANTLR-based C++ parser [Parr and Quong 1995]. Talking to the system designer, we identified several high-level functional aspects of interest:

- *main*: the application's entry point
- *core*: the application's control code
- *logging*: code involved in logging actions
- *GUI*: user interface code
- I/O: code for saving and loading data
- *OpenGL*: rendering code
- *XML*: code for loading 3D models

Each aspect yields an area-of-interest A_i. We now want to see which class participates in which design aspect, and how much. A 'pure' object-oriented design would require each class strongly involved only in one aspect [Lanza and Marinescu 2006]. We quantify the participation degree p_{ij} of each class j in each aspect A_i as its code percentage specific to A_i. For example, an OpenGL class has $p = 0.5$ if it has 50% OpenGL-specific code. The goal is to understand how the identified aspects map to actual classes, *i.e.* whether the code follows the intended design, and whether we have modularity problems.

The entire system is shown in Fig. 8. The legend shows, for each area A_i, the number of classes it contains, the number of classes having missing values for that area's metric p_i (due to the fact that we were unable to reliably estimate the percentage of code involved in each aspect), and the texture used to show the area. We notice several facts. Few classes participate in two aspects, and none take part in three. This indicates a good functional modularity. The only class strongly involved in two aspects is B, part of the *main* and *core* areas. Since B is actually the system's entry point, this strong involvement is not a problem. Class E participates strongly in *core* (red in A_6) and weakly in *GUI* (blue in A_1). E the main window, so its weak involvement in *core* and strong in *GUI* is correct. Class D is strongly I/O-related (A_7), and also part of the core (A_6). However, its code is quite complex, so we were unable to assess how strongly it belongs to the core (missing metric of D in A_6).

Figure 9 shows the same diagram, areas, and metrics as in Fig. 8, with shading added. As for the JPEG decoder example, shading helps users to see quicker which elements are in which areas.

4.3 Adding method-level metrics

Figure 10 shows another class diagram: a part of our own UML visualizer. We show two functional areas: classes involved in visualization (A_2), and the class hierarchy modeling a UML graphical element, or glyph (A_1). Colors show degrees of participation in the two aspects. Since our metric-rendering does not draw on classes, we can show an additional metric: the lines-of-code (LOC) for all class methods. For this, we use a simple adaptation of the well-known table lens technique [Rao and Card 1994]: method metrics are drawn with colored horizontal bars, scaled to show the metric values. Long bars indicate large methods. Methods are sorted in decreasing LOC from top to bottom within each class. This effectively shows the size distribution of all methods, and correlates it with the participation of each class in the two AOIs.

Using this table lens-like technique to show method-level metrics has several advantages. First and foremost, the table lens scales well even for classes having tens or hundreds of methods, as each table row (*i.e.* class method) can be rendered as small as one pixel line. Second, by using the same scaling factors and sorting order for all classes of a given diagram, we can easily compare the range and distribution of a given metric over an entire system. Placing the table lens renderings within each element frame lets us correlate metric values with system structure. Finally, a few different

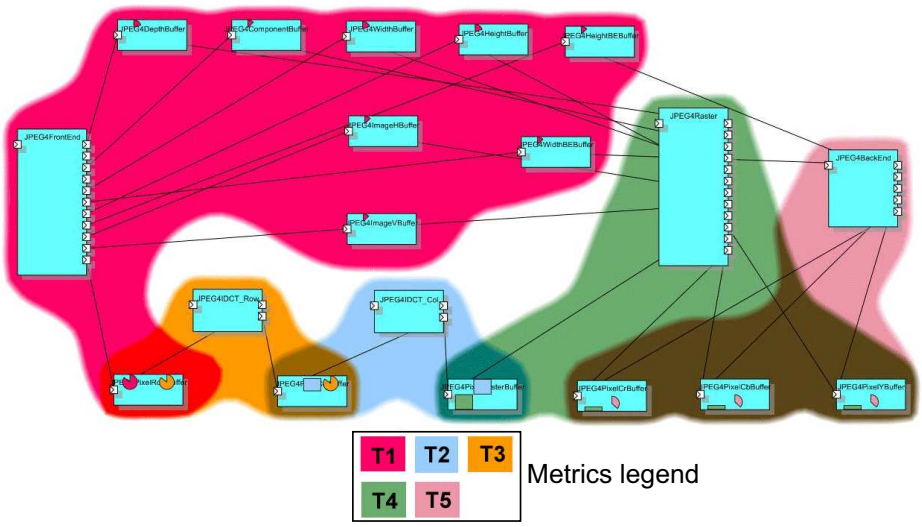


Figure 5: *JPEG decoder architecture. Icons show the memory usage metric μ_{mem} over five tasks. Areas show the tasks. The metric legend shows the placement of metric icons within each component. Although this figure is quite large, it is hard to correlate metric values and areas*

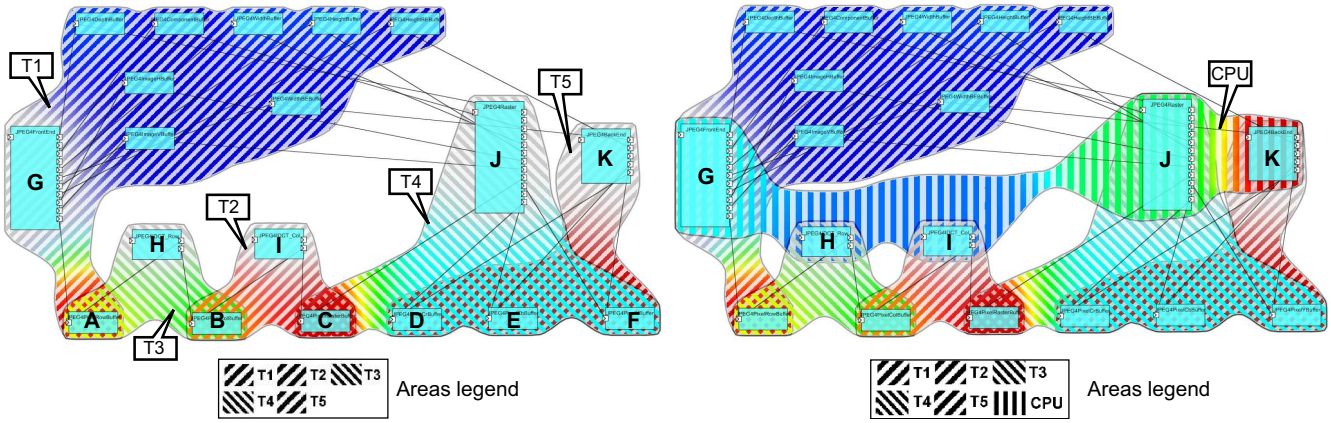


Figure 6: *JPEG decoder architecture. Left: 5 tasks with memory usage metric. Right: a sixth task and a second metric is added (CPU usage)*

method-level metrics can be shown simultaneously, by adding several table lens renderings along each other within each class (this example is not shown here for lack of space).

We can use Fig. 10 to understand how code complexity relates to system structure, to predict potential maintenance hot-spots. First, we see that area A_1 contains a class hierarchy, rooted at A, which is the glyph common interface. A_1 is entirely contained inA_2, which is desirable, as glyphs are visualization objects. All glyph classes in A_1 have the same number of methods and similar bar graphs, *i.e.* similar LOC distributions for their methods. This confirms a desired property: all glyph subclasses should use the same coding pattern. At closer code investigation, this was confirmed. Secondly, we notice that class C, although in the visualization area A_2, has no metric here (is gray). C is also the root of a small class hierarchy. This indicates a *mix-in* class: its code cannot be readily classified as visualization, but it roots several visualization classes, so it is classified as visualization-related. The reason for the mix-in is clear when looking at the class name: C is a C++ STL container (`set`), so its two visualization subclasses inherit implementation rather than interface.

The classes having the largest methods (longest bars) have also the most methods: B, D, E. Stronger, the largest class B has also the largest methods. This suggests a 'God class' pattern [Lanza and Marinescu 2006]. Code examination confirmed this: B contains a (complex) part of the system's data model. Correlating the methods' LOC metric with the areas, we also see that D and E are the largest visualization classes, but the most complex class (B) is located outside these areas. Hence, we identified three potential maintenance hot-spots, two in the visualization subsystem (D,E) and one outside (B). In contrast, the glyph subsystem (area A_1) contains only simple, small, similar-pattern classes, hence should be much easier to maintain.

4.4 Informal User Feedback

We have conducted several informal evaluation studies of our proposed multivariate metric visualization technique.

Our main aim is to compare the effectiveness and acceptance of the new texture-based technique as opposed to classical icon-based techniques. We compared our new method against [Termeer et al. 2005] as both methods are implemented within the same UML visualization tool, so we can share the same user interface, input file formats, and visual look-and-feel. Moreover, we had a relatively large base of users already familiar with this UML tool, in the framework of a 2-year industry-academic cooperation project [Trust4All 2005]. The user base includes around 10 professional software engineers involved in creating UML architecture diagrams, such as the JPEG decoder (Sec. 4.1), and computing quality metrics on

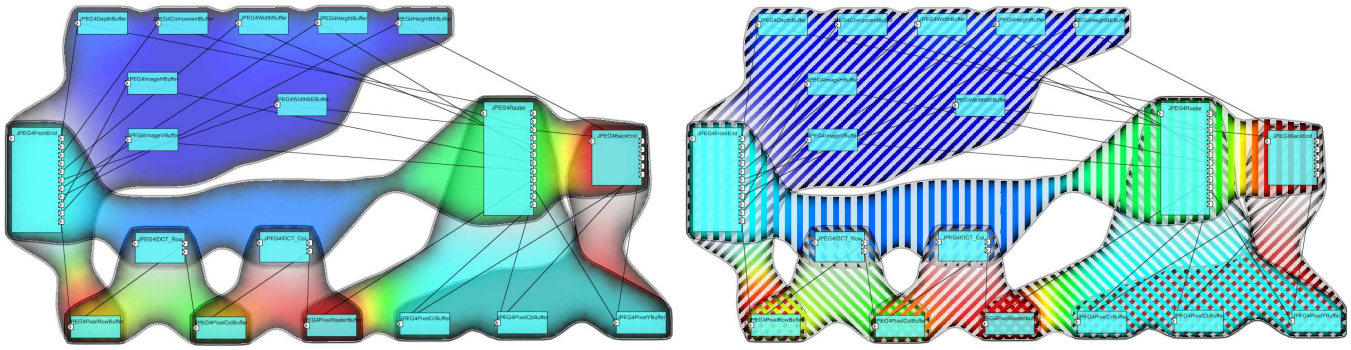


Figure 7: *JPEG decoder architecture. Left: shaded AOIs. Right: Adding shading to textured AOIs (compare with Fig. 6 right)*

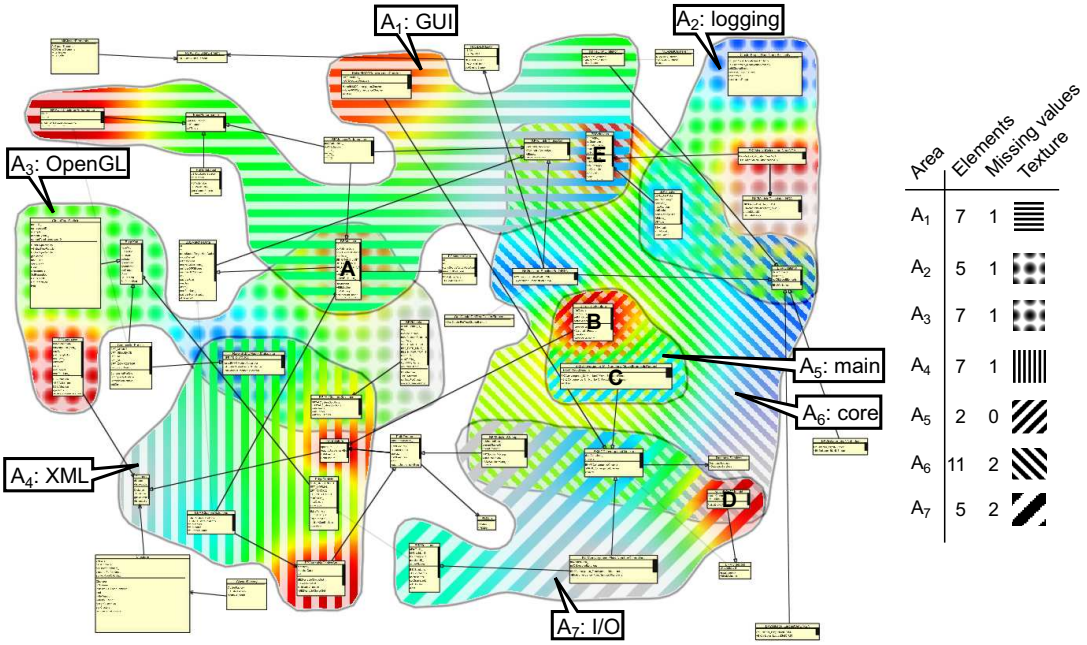


Figure 8: *Large UML class diagram with 7 areas and over 50 classes. Metrics show the participation of classes in two aspects*

them.

We asked the designers to utilize both the icon-based and texture-based metric visualizations to present their own work (diagrams and metrics) in around 10 project meetings of around 20 participants over a period of about 1 year. In such presentations, important goals are to show metric-metric and metric-structure correlations, as described in the previous sections. We silently observed the presentations and gathered off-line feedback from presenters and participants. Overall, there was a strong positive feedback about using areas of interest: they are quick to understand and effective to show software aspects. Metric icons were accepted only when showing a *single* metric, possibly over several areas. Texture-based metric visualizations were seen as more effective and intuitive when correlating *several* metrics. The overlap of more than three metric textures was, however, hard to understand. In such cases, presenters would switch metrics on and off to show only three metrics simultaneously. Interestingly, color smoothing was not seen as a problem, even though it generates colors between the diagram elements which do not correspond to actual values in the data. When talking about this issue, we got the impression that users focus predominantly on the colors *close* to the diagram elements *and* use color smoothing as a visual cue to navigate from element to element over a given area, being aware that in-between colors do not represent data.

5 Disscussion

We discuss several aspects of our technique, as follows.

Scalability: we can easily show up to 10 areas of interest, each with its own metric, on diagrams of 20..80 of classes. Larger diagrams occur very rarely in software engineering practice. The Delaunay triangulator and Laplacian filter used are well-known for their fast, subsecond performance on meshes of thousands of triangles. Rendering a metric over an AOI uses a single texture pass over a triangle mesh, which is very fast on any graphics card.

Understandability: The main limitation is the number of distinct areas that can overlap at one given place. Consider the AOIs $A_1 = (A,B,C,D)$, $A_2 = (A,B,C)$ and $A_3 = (A,B,D)$ in Fig. 11, rendered with textures shown in the legend. From the 'woven' texture pattern we believe it is reasonably easy to see which element is in which area and the colors (metric values) at overlaps. The addition of shading (Sec. 3.3 further helps in separating areas with complex overlaps. Yet, adding a fourth overlapping area would make this image hard to understand. Yet, typical software understanding scenarios rarely involve correlating more than 2-3 metrics at the same time.

Obtaining a good pattern mix constrains the texture parameters. All textures should have similar ratios of opaque-to-transparent pixels, so we can 'see through' at all overlaps. Ratios between 40% and

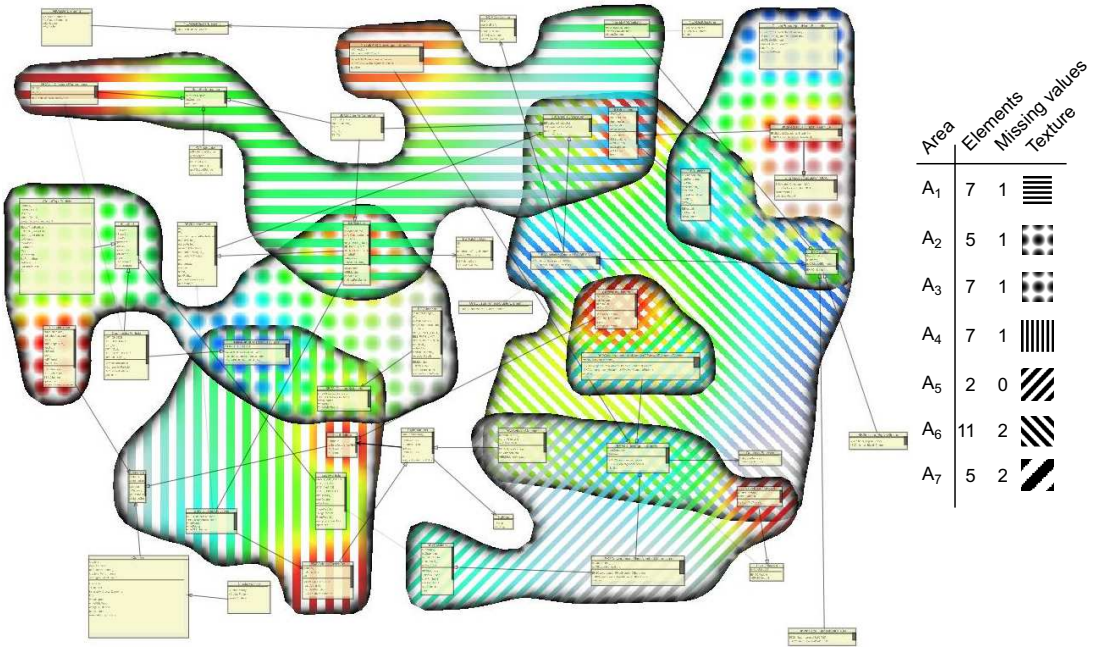


Figure 9: *Visualization of the UML diagram in Fig. 8, now with area shading and half-transparent elements*

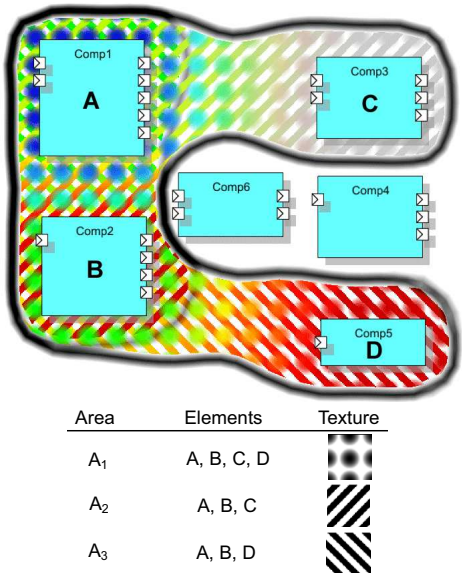


Figure 11: *Complex intersection of three overlapping areas*

60% give good results - lower values yield too sparse textures, on which we cannot see colors or shading; higher values yield occlusion at overlaps, so we cannot see more than one texture. Patterns must be chosen so that the overlap of *any* $n-1$ patterns looks different from the n^{th} pattern, n being the number of overlapping areas. The texture set used here gives good results for $n \leq 3$, as shown in a different application [Voinea and Telea 2006]. Finally, the frequency range (related to the pattern stripe thickness and circle radius) is important. Too thin patterns are hard to distinguish at overlaps; too thick patterns do not let the eye smoothly switch between areas at overlaps. We found an empirically good pattern size in the range of 10..20 pixels (Fig. 2).

Related methods: To our knowledge, there is only one other software visualization that uses textures to show numeric metric values [Holten et al. 2005]. Our method differs from this as follows. Holten *et al* encode two metrics in the texture frequency and luminance, and use a treemap layout, so their areas are rectangular, cannot overlap, and always contain a single element. We smoothly interpolate metrics over arbitrarily-shaped, overlapping areas. We use a fixed texture-set, use opacity to allow overlaps, and encode metric values in hue and metric availability in saturation. Finally, we use luminance to pseudo-shade the areas to visually emphasize contours rather than encoding data. This is conceptually similar to the cushion treemaps used by Holten *et al*, but generalizes to complex-shaped, overlapping, areas.

The colormap choice is very important. Here, we use only a simple blue-to-red continuous colormap, for simplicity and conciseness. However, better choices are available, such as other hue gradients or discrete few-hue colormaps. Such issues need to be further investigated. A second discussion point is our choice to interpolate colors. As mentioned, this creates colors between elements which do not reflect actual values. However, we believe this is acceptable since users are fully aware that there are no data values except on the diagram elements, and smooth colors help following the contents of a given area as opposed to hard color boundaries between elements. Also, using a (discrete) few-hue colormap would considerably alleviate this problem as there would be less, or no, different hues created between the ones in the colormap. Still, a rigorous user evaluation of the effectiveness and/or limitations of color interpolation is still needed.

Finally, using full-saturation hues on the areas is sometimes seen as distracting. Luckily, this is easy to tune: we provide a global opacity control that allows users to set the overall opacity of all AOIs, thus smoothly navigating between 'bare' diagrams and diagrams with full-saturation textured areas. In practice, using a global area opacity of 0.4..0.6 gives good results - the actual value used depending on one's taste and type of color screen.

6 Conclusion

We proposed a method to render a multivariate set of metrics, with potential missing values, on elements of areas of interest on UML diagrams, so that metric-area correlations and distributions of metrics over areas are easy to distinguish. Texture patterns encode dif-

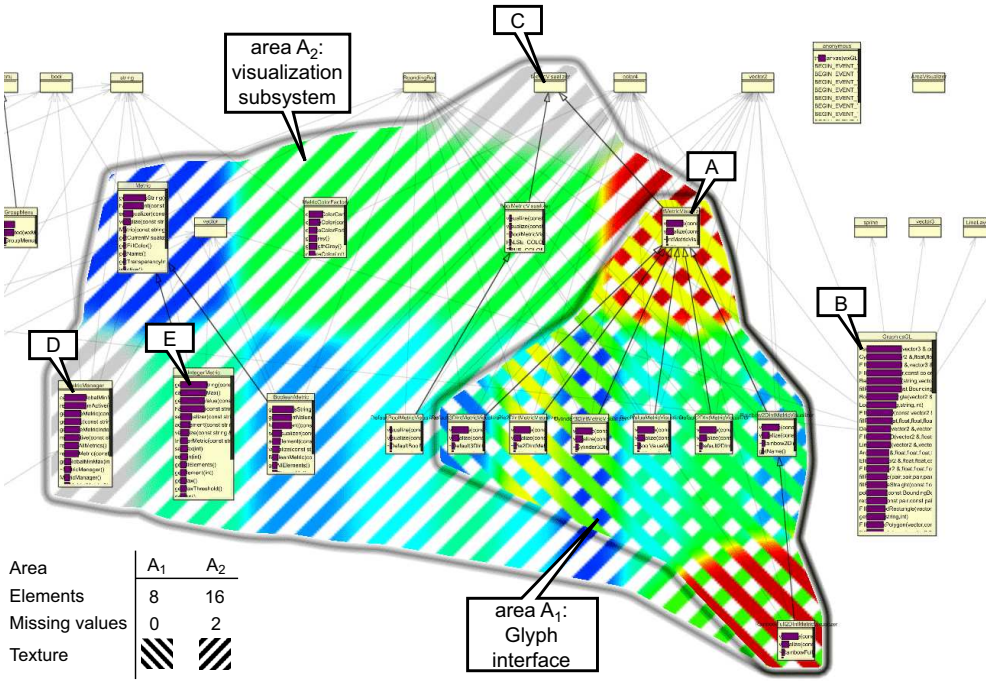


Figure 10: *UML class diagram with two areas, class-level participation metrics, and method-level lines-of-code metrics*

ferent aspects in a diagram and also allow areas to overlap. Additional shading further visually separates complex overlapping areas. The space inside elements can be used to show member-level metrics, using a table lens technique. Although our interest is in software diagrams, our method can be used in other contexts, such as organization diagrams or spatial maps.

We would next like to study how interaction can help understanding the metrics correlation, and also evaluate the practical effectiveness of the proposed methods by means of user studies involving actual software engineers in the industry.

Acknowledgements

We would like to thank Egor Bondarev (TU Eindhoven) for supporting us with the JPEG case study and the ITEA Trust4All consortium for supporting part of our research.

References

BONDAREV, E., CHAUDRON, M., BYELAS, H., AND DE WITH, P. 2006. A toolkit for design and performance analysis of real-time component-based software systems. In *Proc. Intl. Conf. in Software Eng. Advances*, 4–8.

BONDAREV, E., CHAUDRON, M., AND DE KOCK, E. 2007. Exploring performance trade-offs of a JPEG decoder using the DeepCompass framework. In *Proc. Intl. Workshop on Software and Performance*, 153–163.

BYELAS, H., AND TELEA, A. 2006. Visualization of areas of interest on software architecture diagrams. In *Proc. ACM SoftVis*, 105–114.

DA FONTOURA COSTA, L., AND CESAR, R. M. 2004. *Shape Analysis and Classification: Theory and Practice*. CRC Press.

DIEHL, S. 2007. *Software Visualization - Visualizing the Structure, Behaviour, and Evolution of Software*. Springer.

HOLTEN, D., VLIEGEN, R., AND VAN WIJK, J. J. 2005. Visual realism for the visualization of software metrics. In *Proc. VisSoft*, IEEE, 27–32.

LANZA, M., AND MARINESCU, R. 2006. *Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer.

OMG. 2008. *The Unified Modeling Language*. `http://www.uml.org`.

PARR, T., AND QUONG, R. 1995. ANTLR: A predicated-LL(k) parser generator. *Software - Practice and Experience 25*, 7, 789–810.

RAO, R., AND CARD, S. 1994. The table lens: Merging graphical and symbolic representations in an interactive focus+context visualization for tabular information. In *Proc. CHI*, ACM, 222–230.

SHEWCHUK, J. R. 1996. Triangle: Engineering a 2D quality mesh generator and delaunay triangulator. In *Proc. Applied Computational Geometry*, ACM Press, 124–133.

SPENCE, R. 2007. *Information Visualization: Design for Interaction (2^{nd} ed.)*. Prentice Hall.

TERMEER, M., LANGE, C., TELEA, A., AND CHAUDRON, M. 2005. Visual exploration of combined architectural and metric information. In *Proc. VISSOFT*, IEEE Press, 21–26.

TILLEY, S. R., WONG, K., STOREY, M.-A. D., AND MLLER, H. A. 1994. Programmable reverse engineering. *Intl. J. of Software Eng. and Knowledge Eng.*, 501–520.

TRUST4ALL, 2005. The *Trust4All* project. `www.win.tue.nl/trust4all`.

VOINEA, L., AND TELEA, A. 2006. Multiscale and multivariate visualizations of software evolution. In *Proc. SoftVis*, ACM, 115–124.

WUST, J. 2006. SDMETRICS*: The software design metrics tool for UML*. `www.sdmetrics.com`.