Visualization of Metrics and Areas of Interest on Software Architecture Diagrams

PROEFSCHRIFT

ter verkrijging van de doctoraat in de
Wiskunde en Natuurwetenschappen
aan de Rijksuniversiteit Groningen

door

Heorhiy Byelas

Contents

Chapter 1

Introduction

1.1 Trends in software development

In the present days, we notice that software development life-cycle (SDLC) becomes increasingly longer, more complex and more expensive. The reasons for this process are multiple, including, but not being limited to, the following factors: an increase in the required functionality of the target systems, the proliferation of various software technologies and platforms with their many variants and flavors, and a decrease in the expected time-to-market of the constructed software products. On the one hand, the increase in required functionality and number of employed technologies causes an increase of the size and complexity of the artifacts involved in the software construction process, including source code, design blueprints, and documentation content. On the other hand, the decrease in time-to-market causes numerous systems to be developed in a too rapid pace. This creates several types of design and implementation errors, whose removal only further increases cost.

The correlated effects of the increase in software development costs and decrease in time-to-market put a high pressure on software professionals to augment their efficiency and effectiveness in managing the large, complex, and rapidly changing software artifacts.

A typical SDLC, as it is accepted in mature software companies, comprises (at least) the following activities: requirements elicitation, architecture design, detailed design, implementation, testing, deployment, and maintenance. However, the costs of software development are not uniformly distributed over a given product SDLC. Several studies over a period of more than ten years [92, 16] have shown that

- maintenance costs have risen to over 80% of the entire costs of the SDLC
- the largest cost savings can be achieved if errors and suboptimal decisions are identified and corrected in the early design stages

A related observation is that up to 50% of the costs in the SDLC are involved with *understanding* the structure, function and dynamics of the software under construction

[45, 79]. Communication and understanding of the software artifacts is essential in ensuring that each stakeholder can play their role during the design, development, and deployment of a software system [30].

Although there are other important causes of the high cost of the SDLC, such as changeability of requirements, absence of testing environments, scheduling and budgeting difficulties, and improper training of the workforce, we shall limit our analysis in this thesis to the costs involved in understanding the artifacts in a given software product. The main reason for this is that software understanding (and the design of methods to support it) can be studied in more independence on the actual business context in which software is developed, whereas the other abovementioned factors imply a more complex, context-dependent analysis of company-specific social and economical factors.

Combining the above two observations, it seems natural to ask the question whether improved methods and tools can be imagined to assist software engineers to increase their efficiency and effectiveness in understanding software artifacts related to the design process. This is the main goal of this thesis. In the following sections, we shall refine and focus this goal. The remaining chapters describe the way in which we addressed this goal and the obtained results.

1.2 Software understandability

Software has several attributes which make it hard to understand, regardless of the actual context in which it is developed, as follows.

First, software is by excellence *abstract*. It consists of various types of artifacts. In decreasing order of abstraction, these artifacts span several layers, from requirement documents, usually written in plain text; functional specifications, written in either plain text or functional notations; architecture and design documents, written using various notation systems such as UML [65]; and source code, written in programming languages having precise syntax and semantics. Understanding the first types of artifacts mentioned above is particularly difficult, given their often imprecise semantics and not always explcit relation with the lower-level artifacts. For example, an UML diagram constructed in a design process can be interpreted in many ways, and its relation with the underlying implementation code is more often than not hard to determine in an unambiguous manner.

The second element that makes software hard to understand is its *size*. Modern software systems are large. Typical industrial systems consist of hundreds of components, each having hundreds of pages of design documents, and millions of lines of code in their implementation. Since, as already mentioned, the correspondence between system layers is not explicit and unambiguous, understanding such systems involves correlating information from all artifacts in all layers, such as mapping insight extracted from source code onto design documents or extracting actual architectural information from source code and comparing it with the required architecture. Size increases the understanding problem tremendously, especially for weakly modular systems in which hundreds of modules are tightly interlinked in function and structure.

The final factor that increases the difficulty in software understanding is that software is *evolving*. Typical software products have hundreds of releases or versions, developed by tens of programmers over many years. During such periods, changes in the different artifacts, *e.g.* requirements, design documents, and source code, often create incompatibilities which are hard to discover and remove.

From the various types of software artifacts involved in the SDLC, *design* and *architecture* artifacts stand out as being highly abstract, as outlined at the beginning of this section. The most widespread method that allows software practitioners to cope with this abstract nature of design artifacts is to manipulate them via visual representations, such as UML diagrams. Visual representations have several advantages for the design process. First, they capture well the main aspects around which the design process revolves, *i.e.* the structure of entities such as classes, components and packages, and the various types of relations that these entities are involved into, such as dependencies between interfaces and containment of sub-modules in larger modules. Second, visual representations support well the iterative, team-oriented, and exploratory nature of the design process. Although such visual representations are known and used also to manage other types of software artifacts, like code or requirements [22], they are central to the design process.

Regarding visual representation and tasks supported by such a representation, there is no strong differentiation between design and architecture artifacts. Both artifact types share very similar properties and are, in practice, visually represented in conceptually identical ways. Although software design and architecture have given birth to different schools of thought and research directions, we shall not draw a clear cut between these two terms in the remainder of this thesis, and will use the term *design* to capture both meanings. Qualitatively, the only difference we shall associate with these terms is that we regard architectural information (when named as such in the following) as being of a higher level and smaller size, though identical nature from the viewpoint of understanding and visual representation, as compared to detailed design information.

As such, we refine our research question: we are interested in studying how visual representations for design artifacts can be improved to support the effectiveness and efficiency of the software understanding process. In particular, we are interested in supporting the process of combining information from the source code level with design artifacts to improve the understanding process.

1.3 Software design representation

To better define our research question, we introduce the types of software artifacts that are involved in our understanding process.

As we focus on design-related aspects of a software system, we shall review the types of information that are involved at this level. In our research, we are mostly interested in architecture and detailed design, but will also consider source code information, as code and design artifacts are often tightly interrelated.

Design and architectural information consists of a number of so-called *diagrams*. A diagram contains two main types of information: *elements*, or entities, which represent the key artifacts of the design, corresponding to nouns in the design process, such as subsystems, packages, components, and classes; and *relations*, which represent interactions and constraints between elements, corresponding to verbs in the design process, such as depend, inherit, and contain. Together, elements and relations capture the high-level structure of a software system.

In a design, the same elements and relations can (and do) occur in different diagrams. Separate diagrams are constructed to reflect a particular aspect, or function, of a system, as it is impractical and often impossible to manipulate the entire set of elements and relations of a large system on a single diagram. For example, a system can have a structural diagram, a data dependency diagram, a function call diagram, and a deployment diagram, all separated from each other but sharing elements from the same system model.

In many cases, designers identify and use groups of elements of a system design or architecture in their work, without separating these explicitly by constructing separate diagrams. For example, consider all high-performance elements on a system architecture diagram, or all multithreaded elements on a detailed design diagram. A given set of diagrams may have an open set of such groups of elements, and the groups can be intersecting. A key observation is that these groups are highly dynamic: they are created, modified, and deleted during the iterative design process, but live on a given set of diagrams. In contrast, the diagrams change relatively less often, as they capture overall static system properties which are more resilient to change. In the following, we shall call such a group of elements of a design diagram an *area of interest*.

Both elements and relations involved in design diagrams can accommodate several *attributes*. These can be textual annotations, *e.g.* names, but can also be numerical values. In most cases, such numerical values are produced by various analyses of the design, the underlying source code, or the run-time software execution. Such values are called software *metrics*.

Metrics convey quantitative, precise insight in a software system, which complements, but does not replace, the structural insight conveyed by the diagram drawings. Many types of software metrics exist, targeting different aspects of the SDLC, such as maintainability, testability, modularity, complexity, performance, and evolvability [51, 28, 123].

Metrics are of particular use when abstracting large software systems. For example, well-known code-level metrics such as lines of code, fan-in, fan-out, cohesion, coupling, and complexity are used to assess the maintainability and testability of a source code stack without reading the code in detail, and are thus useful in tackling the size aspect of understandability. Metrics can be defined on several levels of detail. If we are talking about an object-oriented system, for example, metrics can be defined on namespaces, classes, relations, class members, up to individual lines of code.

Since there are so many types of information involved in describing a software design, we need effective ways to help engineers understand this information. This is the topic of the next section.

1.4 Software design visualization

Software visualization has been defined as the discipline that makes use of various forms of imagery to provide insight and understanding and to reduce complexity of an existing software system under consideration [22, 30]. Software design visualization is the subdomain of software visualization concerned with the visual presentation of the design-level software representation, *i.e.* the diagrams, entities, relations, and attributes outlined in Section 1.3. As such, software design visualization is a subdomain of information visualization, the discipline concerned with the effective presentation of large amounts of abstract information [118, 90].

Our first research sub-question is how to visually support the presentation and dynamic modification of a large set of areas of interest (AOI) on design diagrams. The envisaged solution should cope well with overlapping areas, and be effective in helping users perform tasks such as assessing which elements are in a given area, and which are the areas that contain a given element.

Neither metrics nor system structure, are, by themselves, enough to understand a software system. In many cases, one has to correlate metric values with system structure to be able to reason about the semantics of the measured quantities. For example, the maximal values of a performance metric need to be correlated with the system elements on which these values occur, and the elements these are related with. Hence, to make the best use of the information contained in software metrics, they should be presented in combination with the system structure. In our case, this means combining metrics with architecture and design structural representations, or diagrams.

Several methods have been proposed to augment design diagrams with software metrics, typically by overlaying or modifying visual attributes such as icons, shading, size, and color in the design diagrams [48, 119, 110, 58, 98]. However, as we shall see in Chapter 2, such methods still have several limitations. Some visualization methods focus on diagram visual representations beyond the UML notations, such as general tree and graph layouts [5, 52] and treemaps [36]. Although this increases scalability, it arguably decreases acceptance, as software engineers are highly trained to use UML diagrams, and also may not have the time needed to learn and use a different visual notation along with their actual UML diagrams in a tight-deadline environment. Other information visualization methods, such as the table lens, are highly effective in presenting large amounts of metrics and metric values [77], and some have been used also in software visualization [104]. However, such representations are not combined with UML (or similar) diagrams, making the correlation of metrics and structure difficult.

Our second research sub-question is how to augment traditional UML diagram visualizations with metric visualizations, such that metric-structure and metric-metric correlations are easy to perform. For this, we are interested in using, or adapting, existing information visualization techniques in the context of software visualization.

Metrics can be defined on areas of interest, much like on the entities and relationships of a standard system diagram. For example, the elements of a system involved in graphics operations, contained in a separate area of interest, may have a numerical metric describing their graphics performance. Our third research sub-question is how to visualize the correlation of metrics and areas of interest. This implies seeing how a given metric changes over the elements involved in a given area of interest, but also how different metrics, defined on different (overlapping) areas, relate to each other.

Concluding, we are interested in creating new methods for visualizing correlations between metrics, areas of interest, and diagrams. This is compactly outlined in Figure 1.1.

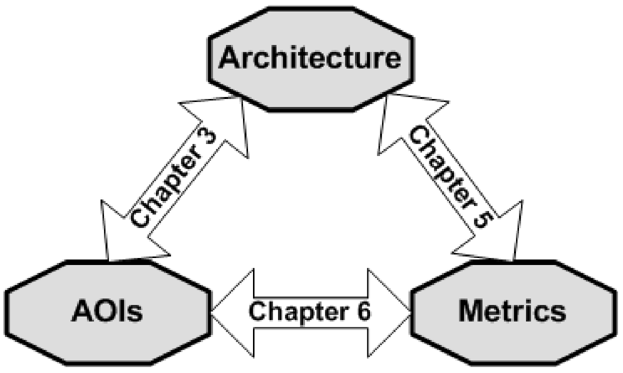


Figure 1.1: Interactions between architecture, AOIs and metrics in this thesis

1.5 Requirements

Throughout out whole research, we are concerned with the following requirements:

- *UML-related:* how to visualize metrics and areas of interest without modifying the layout and look-and-feel of typical UML diagrams;
- *scalability:* how to visualize a large number of metrics, metric values, and areas of interest, defined on large design diagrams;
- *understandability:* how to produce visualizations which are easy to interpret given the typical tasks done by a software designer or architect.

For each specific technique we shall develop, we shall further refine the above requirements, as described in Chapters 3-6.

1.6 Structure of this thesis

The following outlines the structure of this thesis.

In Chapter 2, we review related work in the direction of combining visualizations of software architectures and software metrics. Additionally, we present existing work which is related to our proposed technique for visualizing areas of interest.

In Chapter 3, we introduce two techniques for drawing areas of interest on UML-like software diagrams and discuss their advantages and disadvantages from an algorithmic point of view. The better performing technique is next selected as the basis of the visualizations further presented in this thesis.

Chapter 4 present a user evaluation of the understandability and acceptance of the computer-drawn areas of interest as compared to human-drawn areas on identical diagrams. Secondly, we presents a quantitative comparison of the two types of drawings. The aim of the evaluation is, first, to elicit the visual features of areas of interest which are perceived as relevant by the users, and secondly to measure the degree up to which the computer-generated areas of interest satisfy these desirable properties.

Chapter 5 adds to our software architecture visualizations, enhanced by areas of interest, the presence of class-member level metrics defined on class methods or data members. We present several applications in reverse engineering where combinations of diagrams, metrics and areas of interest are used to gain understanding of aspects of a software system related to maintainability.

Chapter 6 completes our software visualizations by adding the depiction of area-level metrics defined on possibly overlapping areas of interest. We illustrate the applicability of the area-level metrics by a case study performed on a third-party software system.

Chapter 7 presents an evaluation of the developed visualization techniques performed within the framework of a large-scale software engineering project involving both research and industry practitioners. We detail the place and role of visualization in the entire project pipeline and illustrate the added value of visualization by means of an example application.

Chapter 8 discusses the presented visualization techniques from the perspective of the main requirements introduced in this chapter, namely the addition of areas of interest, area-level metrics, and class-member level metrics to UML-like visualizations of software diagrams. The strong points, as well as the limitations, of the proposed techniques are pointed out.

Finally, Chapter 9 concludes this thesis and outlines potential directions for future research.

Chapter 2

Related work

The aim of this thesis, introduced in Chapter 1, is the creation of scalable and understandable UML-like visualizations that combine software structures, software attributes, and areas of interest. In this chapter, we present an overview of work that is related, either in aim or techniques used, to the above goals. Firstly, we give a brief classification of existing software visualization techniques. Next, we present a number of recent, ”state-of-art” visualization methods, which combine the visualization of software structure and software attributes. We next review methods related to the visualization of groups of elements considered on general-purpose diagrams, since several such methods have been created in contexts outside software visualization. Detailed techniques that are specifically related to the material presented in the remainder of this thesis are discussed further in their relevant chapters.

2.1 Introduction

Diagrams are used to show the structure of software since the very beginning of computer programming [22]. Among the first kinds of software diagrams were Jackson diagrams [43], control-flow graphs [34], and structograms [64]. However, with the growth in the size and complexity of software systems, visualization goals moved from detailed algorithm and data structure representations to higher level of abstractions in which diagram entities are classes, components, or even entire subsystems. Along with the visualization of software system structures, done by means of different types of diagrams, different methods have been imagined to bring software attributes, whether numerical or not, in the picture.

This thesis is devoted to visualization of software structure combined with quality attributes represented by metrics. Within this scope, a large part of the work proposed here concerns itself with the visualization of groups of software diagram elements or areas of interest. Hence, the remainder of this chapter is divided in a part over related work on the visualization of combined software structure and attributes (Section 2.2) and a part over related work on visualizing areas of interest in general diagrams (Section 2.3). In both these sections, we discuss existing techniques with an eye on the general requirements related to our research goals stated in Chapter 1, *i.e.* the creation of scalable and understandable UML-like visualizations that combine software structures, software attributes, and areas of interest.

There are many ways to classify the existing body of work in software visualization. Here, we shall use a classification model which is inspired by the model proposed by Marcus *et al.* [58]:

- *Domains* and *tasks* of software visualization: A list of possible domains usually includes software comprehension, maintenance or reverse engineering. We should consider for what specific tasks visualization is needed and who are the stakeholders. Additionally, we need to know the questions we try to answer with the studied visualization.

- *Scope* of visualization: Here, scope refers to what artifacts to visualize. These can be artifacts from any phase of software development life-cycle, including knowledge data, requirements and risks, structural dependencies, source code, and static and dynamic analysis data.

- *Data gathering method*: The data can be collected automatically from the source code or other artifacts, inserted manually by the users, or a combination of both.

- *Interaction* and *representation*: User interaction and navigation consider a number of views which represent the data and navigation between these views. Different properties of software can be shown in a *signle* view or can be present in different *multiple* views or diagrams.

- Visualization *dimensionality*: 2D or 3D representations are usually used for software visualizations.

2.2 Visualizing software structure and attributes

Visualizations that combine software structure and software attributes are arguably among the most ubiquitous types of software visualizations in existence, and among the first being proposed in the history of software visualization [22, 90]. Along the relevant dimensions of a software visualization method introduced in Section 2.1, the chief domain of combined structure-and-attribute visualization is software comprehension, typically within the framework of software maintenance activities. In such contexts, understanding the structure of a potentially large, complex, and relatively unfamiliar software system is best served by visualizing the structure. Moreover, adding attributes such as quality metrics to this picture helps correlating various quantitative insights with structural and architectural insights [51].

Since so many structure-and-attributes visualizations exist, it is relevant to outline their main common features and emphasize any shared strong points or limitations. From our experience, most such visualizations share two design elements, as follows

- *structure:* software structure is typically depicted by using a node-and-link graph metaphor, where nodes are software entities, *e.g.* functions, classes, components, or packages, and links are the relevant (sub)set of considered relations, *e.g.* function calls, data dependencies, associations, or inheritance relations. For example, the well-known UML notation is nothing else than a formalized representation of such a visualization for a specific scope of activities.

- *attributes:* software attributes are usually depicted by mapping them to a visual attribute of the corresponding nodes or links in the structure visualization. Visual attributes that can be used to show software attributes are the position, size, shape, color, texture, lighting, line size, and annotations of diagram elements.

In the following, we present a number of structure-and-attribute visualization designs. Given the highly practical and applied nature of software visualization, we identify each design by an actual software visualization tool. For each such example, we outline its main characteristics as well as emphasize some of its limitations relevant to our research goals stated in Chapter 1.

2.2.1 SHriMP

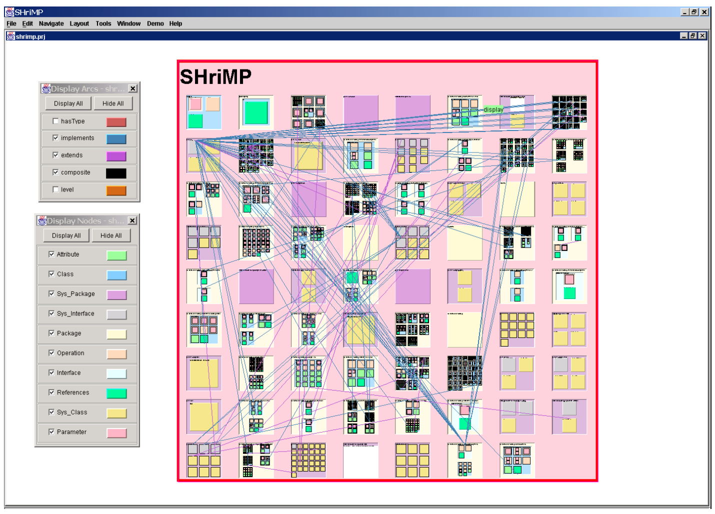


Figure 2.1: A SHriMP view.

The SHriMP (Simple Hierarchical Multi-Perspective) technique provides a customizable and interactive environment for navigating and browsing complex information spaces. Its goal is to combine graphical high-level views with textual lower-level views in order to ease navigation within large and complex software programs. The SHriMP tool attempts to visualize multiple aspects of an architecture through a single view. The primary view in SHriMP uses a zoom interface to explore hierarchical software structures. Essentially, this view shows two types of relations at once:

- *containment* of software elements is shown by visually nesting their box-shaped representations in a containment layout. This nested layout shows parent-child relationships in the system. For example, a Java program hierarchy can include packages, classes and interfaces, attributes and operations.

- *association* of software elements is shown using a traditional node-and-link metaphor. The layout of these relations is typically constrained by the containment layout. Different types of relations can be shown using different colors for their lines in the drawing.

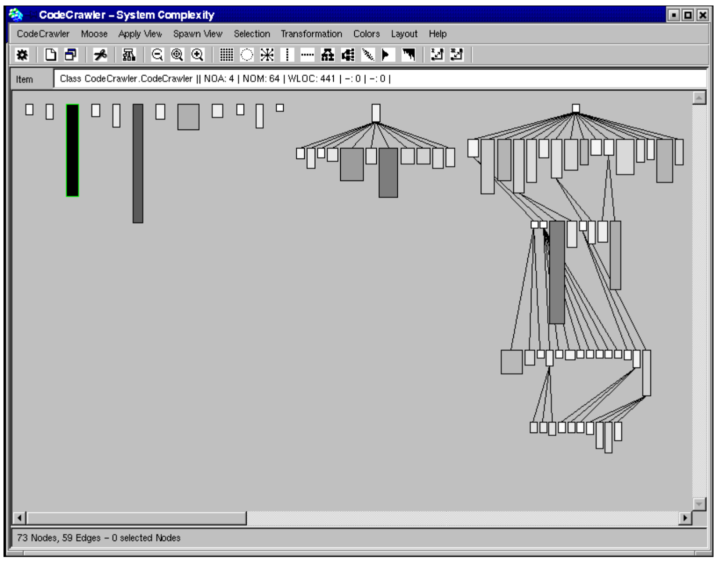
SHriMP is implemented in several software visualization frameworks, among which the one bearing the same name [93], the well-known Rigi reverse engineering framework [110], and the SoftVision framework [103]. These frameworks support tasks such as understanding the structure of large and complex software systems at different levels of detail by allowing the user to visualize the element containment hierarchy at the desired depth, filter specific elements or relations in or out of the picture, and changing visual attributes. Figure 2.1 represents a SHriMP view showing the architecture of the SHriMP program itself.

SHriMP-like techniques favor understanding by exploration, and thus come together with many interaction features, such as fisheye zooming, semantic zooming as well as geometric zooming to serve different purposes of navigation. Geometric zooming allows the user to scale a specific node in the nested graph. Fisheye zooming allows the user to zoom on a particular piece of the software, while simultaneously shrinking the rest of the graph. Semantic zooming chooses a particular view for a node depending on a particular task. Moreover, software attributes such as element and relationship types, but also numerical attributes such as *e.g.* entity sizes (measured in lines of code or a similar metric), can be mapped to the elements' visual attributes such as size and color.

SHriMP gives a good overview of hierarchical structures in a single representation and is reasonably successful in showing the correlation between two types of relations (containment and association) and one or two element and/or relation attributes. However, this method also has some limitations. The combination of nested and node-and-link layouts used by SHriMP can sometimes create cluttered images, especially for medium-size or large systems. Such cluttering is generally absent in typical UML diagrams which are carefully laid out to reduce edge-edge and edge-element crossings. The nested layout also trades off the possibility of showing information for the non-leaf elements of the containment hierarchy within the elements themselves, as this space is used to show the children, much like a treemap metaphor. A related problem is that showing more than two or three attributes per element, such as the signatures of the methods of a class in a UML diagram, or several software metrics computed in that element, is relatively hard.

2.2.2 CodeCrawler

CodeCrawler is a lightweight software visualization framework that aims at adding software metric information to software structure visualizations [51, 48, 49]. Similar to Rigi and SoftVision, CodeCrawler is mainly targeted at static software visualization in the context of reverse engineering and program understanding. CodeCrawler is used to implement several software exploration techniques as *class blueprints* [23] and *polymetric views* [50], which are compact ways of summarizing the correlations of a number of software metrics over medium-sized and large software systems. Also, similar to the other software visualization frameworks cited above, CodeCrawler can be used to visualizing generic structures and data attributes, *i.e.* which do not necessarily come from the field of software reverse engineering.



Class CodeCrawler.CodeCrawler || NOA: 4 | NOM: 64 | WLOC: 441 | -: 0 | -: 0 |

73 Nodes, 59 Edges – 0 selected Nodes

Figure 2.2: A CodeCrawler main window.

Figure 2.2 shows a typical visualization created with CodeCrawler. The visualized system in this case is CodeCrawler itself. CodeCrawler offers multiple correlated views to explore a software system, as follows

- a *coarse-grained view* shows an overview of the system
- several *fine-grained views* show additional details in selected parts from the coarse-grained view.
- a *coupling view* shows coupling between modules in a given software architecture
- an *evolutionary view* shows changes of a given architecture over time using an evolution matrix. Each column of the matrix represents a version of the software, while each row represents the different versions of the same class.

From our perspective, we are mainly interested in analyzing the coarse-grained and fine-grained views of CodeCrawler. As visible from Figures 2.2 and 2.3, CodeCrawler uses similar node-link structure visualization strategies to the SHriMP tools discussed earlier. Various layouts are possible, such as trees, directed graphs, and a combination of nested and tree layouts quite similar to the idea of SHriMP. This is visible in Figure 2.3 which shows a CodeCrawler fine-grained view called the *class blueprint* [23] in the architecture. Class blueprints support the reconstruction of the logical flow of method calls by decomposing a class into layers. The methods and attributes are shown using boxes and positioned according to the layer they have been assigned to.

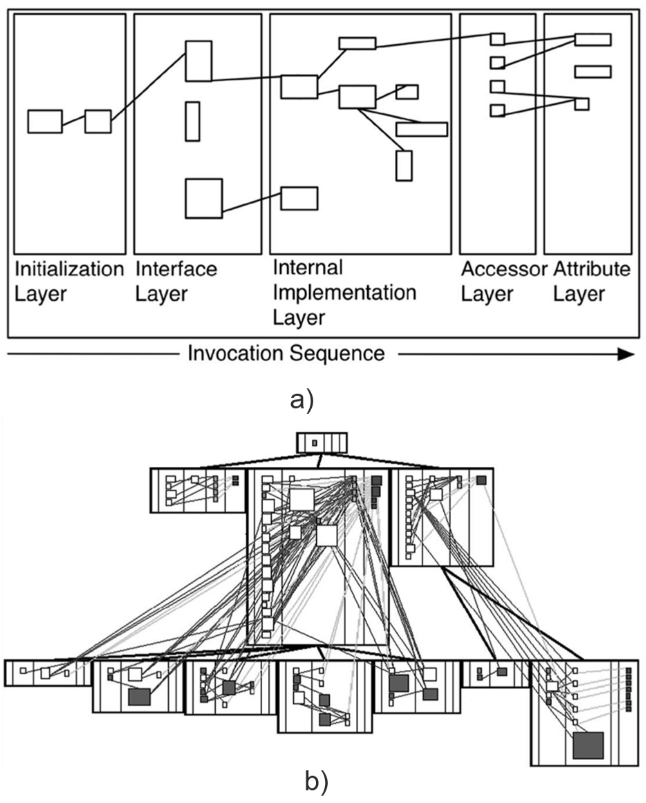


Figure 2.3: (a) a class blueprint structure. (b) a class blueprint visualization example.

Similar to the SHriMP-like tools discussed earlier, CodeCrawler offers standard visualization interaction and navigation techniques to favor exploration, such as linked views, different types of zooming, and the possibility to interactively select and layout subsystems of interest.

CodeCrawler lays a strong emphasis on understanding object-oriented systems. As such, it is integrated with third-party tools that compute such metrics from source code [60, 62, 29]. Several tens of different object-oriented metrics have been visualized in this way with CodeCrawler, *e.g.* lines of code, lines of comment code, number of methods per class, class cohesion and coupling, and cyclomatic complexity [51]. Metrics are visualized by mapping them to the color, height, width, and position of the element box icons (Figure 2.4). An interesting design decision is to use the elements' height and width dimensions independently to show two different software metrics and their possible correlations. Indeed, artifacts having two large metric values will show up as large balanced aspect-ratio rectangles, while artifacts where the two metric values differ significantly will show up as thin and long, or thick and short, rectangles. However, showing more than 2..3 attributes per entity becomes difficult, since the only parameterizable visual dimensions are the elements' sizes and colors.

The visualized entities represent concrete and non-concrete software artifacts. Concrete artifacts can be localized in the source code and include classes, methods, functions, or packages, whereas non-concrete artifacts cannot be localized within the source code, but represent often abstractions in the head of the developers. Examples for non-concrete artifacts are groups of classes, subsystems, or aspects, which exists from a design perspective, but do not have direct equivalents in the code. As such, CodeCrawler is able, in principle, to show the so-called areas of interest introduced in Section refc:introduction:s:representations. However, since all CodeCrawler entities are drawn as boxes enclosing, or connected to, other boxes, visualizing a complex set of several partially overlapping areas of interest using such a method is not a scalable or easily understandable option.

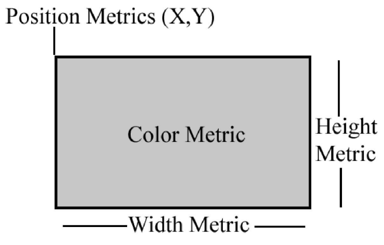


Figure 2.4: A graphical representation of classes and metrics.

2.2.3 CodeCity

In the quest for more visual dimensions to use to show additional attributes, several researchers have proposed the use of the third spatial dimension. For example, CodeCity [119] proposes a software visualization metaphor that extends the basic ideas of SHriMP and CodeCrawler: the structure of the software is represented using a two-dimensional layout. The third dimension, orthogonal to the layout, is used to represent a scalar software metric defined on the software entities. Overall, the global impression is that of a city map populated by skyscrapers, the classes being buildings located in districts representing the packages where the classes are defined. Using the third dimension allows showing one additional metric. For example, CodeCity was used to show object-oriented Java software: the number of methods of a class is mapped to the class building's height; the number of data attributes is mapped on the building's base size; and the nesting level of the class within a package is mapped on the district's color saturation.

CodeCity propose two levels of visualization:

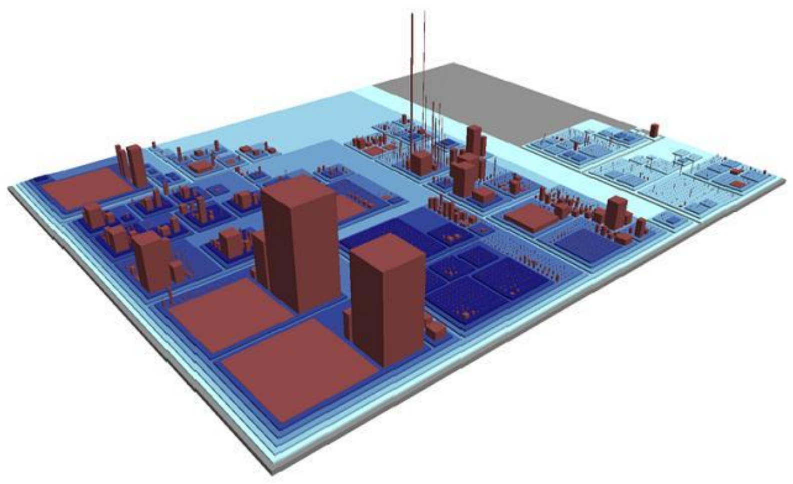


Figure 2.5: An example of a CodeCity visualization

- *coarse-grained*: classes are represented as monolithic blocks, omitting internal details (shown in Figure 2.5)

- *fine-grained*: class methods are rendered as bricks that constitute the body of the building corresponding to the class (shown in Figure 2.6)

Besides software structure and metrics, CodeCity shows its evolution too using several techniques. For example, Figure 2.6 shows the evolution of a class at a fine-grained level. In this image, we can easily see the growth of the class in a number of methods. Missing bricks of the building show which methods were removed and when this happened.

The CodeCity authors claim that it can easily show thousands of classes whereas it still allows to identify outliers, like two big buildings in Figure 2.5. A city metaphor offers a clear notion of locality, thus supports spatial orientation. Still, this type of visualization has several limitations. First and foremost, 3D software visualizations are prone to several well-known problems, such as occlusion, difficulty in choosing a viewpoint, and foreshortening effects causing difficulty in interpreting sizes correctly. This is especially critical when visualizing large, complex systems having hundreds of classes each with tens of methods. In such a case, seeing all the details of all 'buildings' is clearly not possible. Second, CodeCity does not show relations between the software elements, as CodeCrawler and SHriMP do. Although it is possible to add such relations, *e.g.* using 3D splines connecting the buildings or drawing them on the back of the city's surface, this would create too much clutter in combination with the third dimension. Finally, since different metrics are mapped to different visual dimensions, such as building heights, base areas, and colors, it is hard to compare such metrics among themselves.

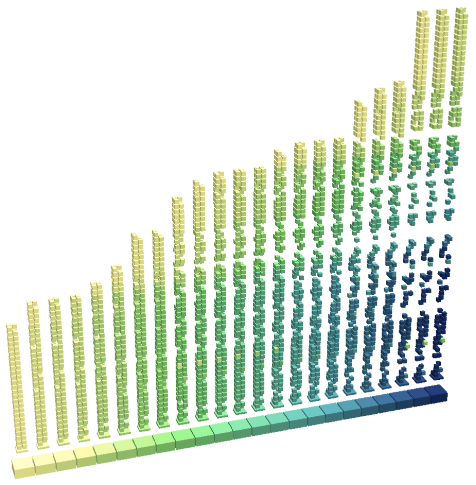


Figure 2.6: An example of the fine-grained CodeCity visualization with a notion of time.

2.2.4 Attribute-enhanced 3D city metaphor

Panas *et al.* refine the 3D city metaphor proposed in CodeCity by adding a supplementary level of realism that helps in depicting additional attributes. The scope of this visualization is understanding cost-related information of software production during software maintenance, rather than software metrics obtained from static analysis as in CodeCity. The focus here is on the business context of software production, *e.g.* getting insight in the costs and efforts required to design, develop, test and maintain a software product. This visualization aims to help in indicating the system hot spots and high cost areas. The stakeholders here are not only the system developers or maintainers, but also project managers, who are interested in business related information of a software system.

Although the visualization discussed here shares the same basic idea of a 3D city metaphor with CodeCity, we see some enhancements and differences (Figure 2.7). The components under modifications are indicated in yellow with the respective names, the execution hot spots are are surrounded by animated fire, various system aspects [2] are shown in specific colors, buildings with flashes indicate frequent component modifications, and so on. The graphic detail in representing the buildings that show the software elements is much higher than in CodeCity. Various textures are used both for the animations that outline specific attribute values as for the basic appearance of the different buildings themselves.

Using a realistic 3D city metaphor for showing business-related information about software system can be useful in very high level system development analysis, where technical and business aspects intersect and should be discussed together. The realistic, animated effects are quite suggestive in attracting the attention of the user to specific values and events, which makes the visualization easier to follow, especially for non-technical users. The use of high-detail textures and shading is a promising idea, which we shall exploit ourselves in the creation of our own structure-and-metric visualizations (see

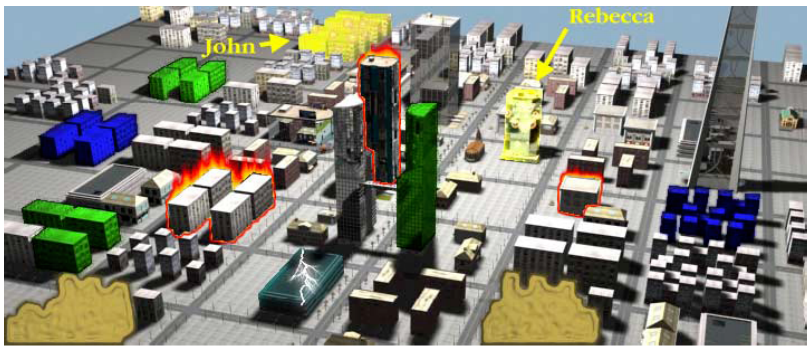


Figure 2.7: A 3D City example combining software structure and business-related software information

e.g. Chapters 3 and 6).

2.2.5 MetricView

MetricView [107] is a software visualization and exploration tool that combines traditional UML diagram visualization with metrics visualization. In contrast to all visualizations discussed so far, MetricView uses existing class, sequence, state, use case, and collaboration UML diagrams as a basis for software structure visualization. This has the important advantage of presenting structure in well-known terms to many software engineers, who are expected to easily understand the multiple views given by UML diagrams. Apart from the diagrams, MetricView supports the visualization of metrics defined on UML diagram elements. Metrics can have boolean and numeric values. Any UML model element can have any number of metrics, such as obtained from reverse engineering or reverse architecting [51]. Metrics are visualized as icons, drawn atop of the UML elements for which the respective metrics are available. For example, the icons for integer values can be 2D rectangles, colored using different colormaps, 2D height bars, circles, and pies, 3D bars or cylinders, and more. Boolean metrics are visualized using a 2D checkbox-like icon.

Figure 2.8 shows an example of UML class diagram with several metrics rendered using 2D icons visualized with MetricView. Although the 2D layout used in this diagram is arguably the most familiar one to software engineers using MetricView, the tool is not limited to two-dimensional visualizations. For example, Figure 2.9 shows the same diagram as in the previous figure, now using 3D icons. 2D icons use color, width and height in the 2D plane, or shape (*e.g.* for the pie) or appearance (*e.g.* for the checkbox) to show the data attributes. 3D icons use color and the height orthogonal to the 2D plane to show data. As such, MetricView is conceptually similar to the ideas behind CodeCity or the work of Panas *et al.* discussed earlier: Structure is displayed using a 2D diagram, while metrics are shown using the third dimension. Metrics are correlated with structure by means of containment of the metric icons within their respective diagram elements.

MetricView has a number of quite effective design elements which are relevant for

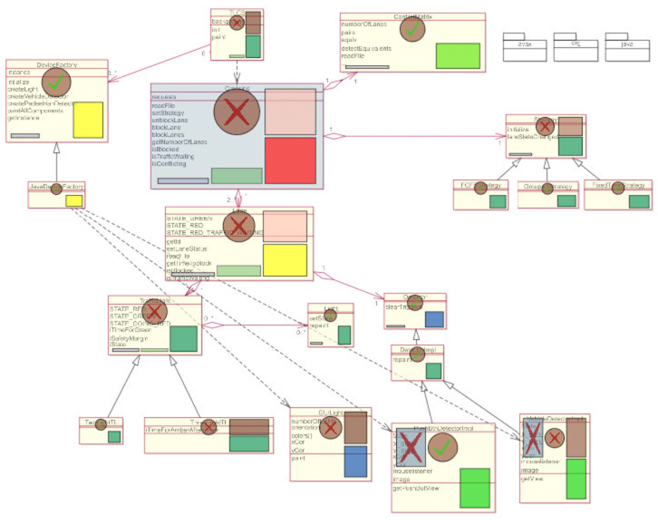


Figure 2.8: An example of 2D UML class diagram visualized with MetricView.

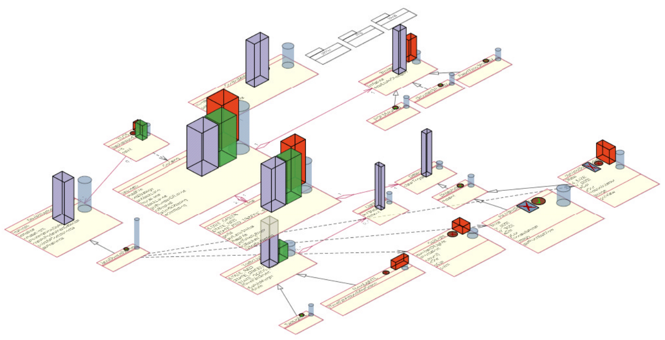


Figure 2.9: An example of 3D UML class diagram visualized with MetricView.

our goals. First and foremost, it promotes a UML-like visualization. If desired, users can interactively tune, or even completely switch off, the opacity of all metric icons, in which case the visualization becomes a classical UML diagram. This allows a smooth transition from the familiar UML diagrams showing just structure to UML diagrams enhanced with software metrics. Also, the layout of the UML diagram is not affected, in any way, by the metrics visualization. Metric icons simply take the space given by the UML diagram layout, which immediately allows users to import existing third-party diagrams with no effort. In other words, metric information is added to diagrams in a non-intrusive way. This is important, as users keep their ’mental map’ of diagrams they are accustomed with. In contrast, the visualizations discussed earlier typically create their own layouts from the input structural data. Small changes in the input data may cause relatively large visual changes in the layout, which makes following such visualizations more difficult. This is indirectly related to a difference in focus between MetricView and the tools discussed earlier: While the former mainly focuses on *design*, where diagrams (and thus, their layouts) are constructed by users, the latter mainly focus on *maintenance*, where visualizations are constructed from reverse-engineered data often acquired from unnknown systems.

MetricView offers a wide range of fine-grained visualization customization options. These allow users to specify which metrics to display, how to arrange (layout) them within each diagram element, and which graphical shapes, colors, sizes, and transparency options to use for the metrics. MetricView is very effective in showing up to 10..15 metrics defined on each class with little clutter, hence is quite scalable in this respect. However, it has no provisions to show method-level metrics, areas of interest, or area-level metrics.

All in all, the design promoted by MetricView combines the closest the requirements stated in Chapter 1 for our research goal: Show, using a UML-like metaphor, a combination of software structure and metrics defined at various levels of detail, as well as additional structures such as areas of interest, potentially having their own metrics. As such, we chose most design elements in MetricView as the basis for our own research, *e.g.* the classical 2D UML structure visualization; the use of texturing and blending to subtly add attribute information to an existing structural picture; and the dominant 2D visualization flavor and look-and-feel of the tool, which strongly reminds of a typical UML diagram editor. Further in this thesis, we shall show how to extend these design elements to add visualizations of metrics defined on class members (Chapter 5) and visualizations of areas of interest and area-level metrics (Chapter 6). MetricView was used as a basis for the AreaView tool presented in Chapter 7 which was used to test our additional visualizations in various usage contexts, including industrial applications.

2.3 Visualization of areas of interest

A particular goal of our research, as stated in Chapter 1, is to create visualizations able to show *areas of interest*, or groups of software system structure elements. Additionally, these areas of interest may have software metrics defined for their elements, which have to be visualized too. In this section, we present research related to the visualization of groups of elements.

2.3.1 Preliminaries

Before we proceed, let us make some important distinctions. Groups of elements can be visualized many different ways. By their own nature, groups of elements are defined on an element set, which is the ’main’ target of the visualization, following the terminology of Marcus *et al.* [58]. In our context, this element set is an entity-relationship dataset, or a graph. For example, all security-related classes in a class diagram constitute a group of elements, which needs to be visualized in the context of the diagram itself. As such, a first dimension of the choice space is whether we want to visualize the groups of elements *within* the main visualization of this entity-relationship dataset, or *outside* it. The second option is the simplest, and can be achieved in many ways, *e.g.* by listing the elements contained in each area of interest in a separate tabular view. However, this option does not allow one to easily correlate the areas of interest with the main visualization, *e.g.* to answer questions like 'which are all areas containing a given element' or 'which are all elements in a given area'. The first option attempts to visualize the areas of interest in the same view as the main entity-relationship dataset. This is a preferred option, as correlating areas with elements, or areas between themselves, should be easier. Therefore, we choose this option in our research.

A second dimension of the choice space is whether the layout of the entity-relationship dataset on which areas of interest are defined is *fixed* or *free*. Having a free layout allows the visualization to arrange the elements in ways that simplify the visual construction of the areas of interest. This is, by far, the most used technique so far in information visualization and software visualization applications, as we shall see next. However, as already mentioned, our desire is to keep the layout of a given software diagram fixed when adding metric and area-of-interest visualizations to it, so we cannot take this way. Having a fixed layout, however, diminishes the freedom for visual construction of areas of interest. We shall present in Chapter 3 a method that constructs visualizations of areas of interest on fixed diagram layouts.

A third dimension of the choice space is how to display the fact that an element is within an area of interest. Several options exist. First, one can mark the element using icons or colors that allow the identification of the area(s) it is contained within (each area will have a particular color or icon assigned). This option is directly supported in several InfoVis and SoftVis toolkits such as Prefuse [70], Rigi [110] and the InfoVis toolkit [27]. In particular, the MetricView tool also supports this option using icons, as discussed further in Chapter 3. However, this option has the marked drawback that it is hard to see which are all areas containing an element, as areas are not drawn explicitly. A second solution is to draw the containment relations between elements and areas explicitly using a node-link metaphor: Both elements and areas are represented as nodes, and edges are drawn from areas to all elements they contain. The resulting visualizations can be displayed using standard graph layout algorithms such as *e.g.* provided by the GraphViz toolkit [5]. If there are not many elements contained in the same time in different areas, the results are suggestive, since graph layouts such as spring embedders will naturally arrange the elements around the nodes representing area(s) in which these are contained (see *e.g.* [100]). However, this works well only if there are few intersections between areas and also if we are allowed to construct an own layout of the dataset rather than use a given one.

Another solution of displaying the containment relations between elements and areas is to render them as visual containment. That is, an element contained in an area will be drawn inside the visual representation of that area. The element-in-area containment relations can be rendered in various ways, such as using treemaps or nested layouts such as the SHriMP layout discussed earlier in Section 2.2.1. Identifying all elements within an area is now very easy. However, using rectangular containers such as treemaps or SHriMP almost always implies that we have to re-layout a given diagram to construct the visual nesting of elements. Moreover, such solutions do not work well if there are many intersections between areas of interest, since rectangular containers are best suited for visualizing strict hierarchies, as shown later in Chapter 3 by means of an example.

The remaining solution is to show visual containment by means of other shapes than rectangles. A classical solution of this type are the traditional renderings of mathematical Venn-Euler set diagrams, where elements are represented as nodes, and areas are represented as smooth, curved shapes surrounding the elements they contain. This type of solution has many advantages. First, we can use a given layout, subject to the geometric freedom of the shapes used to draw the areas. Second, the element-in-area containment is clearly shown as visual nesting. Third, it is easy to see which are all areas containing a given element. Last but not least, Venn-Euler diagrams are intuitive for most users.

Given the advantages of Venn-Euler-like renderings of areas of interest, we shall use this idea as the basis of our further construction of areas of interest for software diagrams. In the following, we discuss a number of visualizations that are related to this type of method.

2.3.2 Visualization of interconnected social groups

Social networks can be separated in groups, such as friends, families or people involved in the same project. Theron *et al.* [108] developed a technique which shows such groups by wrapping them by smooth shapes. The main requirements to area construction are to surround elements which belong to a group and to show clearly areas' intersections. The layout of elements can be changed to improve the final result. Figure 2.10 shows a conceptual example of three social groups. People in the group are represented by different shapes, *e.g.* rounds and squares. Groups of elements are surrounded by contours and half-transparent areas. The idea of area construction is similar to the one presented in Sec. 3.3. However, while the image shown in igure 2.10 is only a conceptual sketch in [108], we present in Chapter 3 an actual algorithm able to automatically compute such shapes from a set specification of their contents.

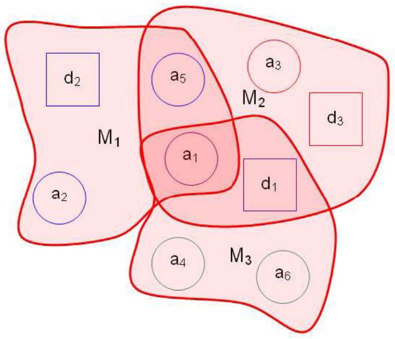


Figure 2.10: The idea of social group visualization.

Figure 2.11 shows an shapshot of the implementation of the social group visualization. In this figure, there are two intersecting areas which represent the most awarded movies (yellow) and the movies that earned more money (green). The nodes in the areas are actors who take part in these movies. In the implementation of this technique, a lot of attention is dedicated to highlighting a particular node of interest and showing its details, as well as showing an overview of the social network.

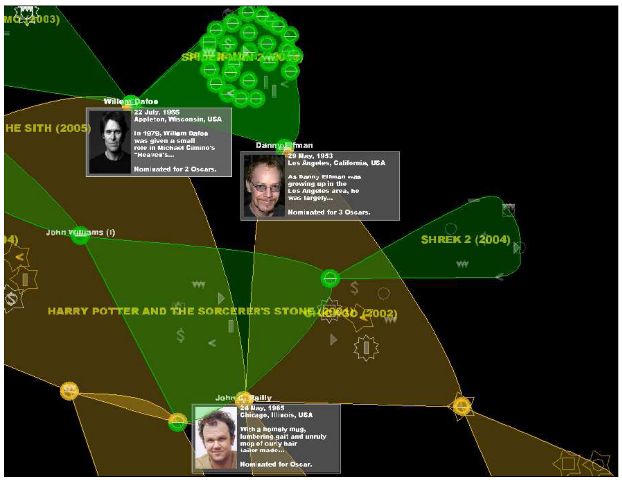


Figure 2.11: An implementation of social group visualization.

2.3.3 Clustered graph layouts

Balzer *et al.* [6] propose to use implicit surfaces for the visually simplified representation of vertex clusters and so-called edge bundles for the simplified visualization of large and complex graphs. A major problem of graph visualization is the limitation in the number of vertices and edges that can be visualized. If a graph contains more than a few hundreds of vertices and edges, it can result in an incomprehensible representation with many overlappings and occlusions, which makes a study of individual elements almost impossible. Furthermore, a large amount of graph elements can cause a drop in the visualization performance and interactivity.

The *clustered graph layouts* solution is based on the substitution of a complex object by a simplified object or the substitution of a group of objects by a single one. It focuses on the spatial grouping of vertices, whereas the edge routing or the minimization of edge crossing is of less importance. Edge renderings are constrained by the vertex positions in the graph and the simplification of the graph. This means that an edge is visualized by a direct connection between two vertices and all edges between two clusters of vertices are represented as one aggregated edge that is also visualized by a direct connection (line) between the clusters. An important role for this technique plays a chooice of the level-of-detail for clustered graph layouts visualization and a viewpoint of the user. The changes between the detail levels should be continuous to preserve the mental map of the user and to enable the comprehension of abstractions. Figure 2.12 shows an example of a graph visualized with the clustered graph layouts technique. A similar approach is presented by Gross *et al* in [91]. A related technique from the perspective of graph clustering is presented by Van Ham *et al.* [112], with the difference that clusters are, in the latter, rendered using spheres, whereas Balzer *et al.* use the more flexible implicit surfaces, discussed next.

The cluster representations in [6] are visualized using an *implicit surface technique* [9].

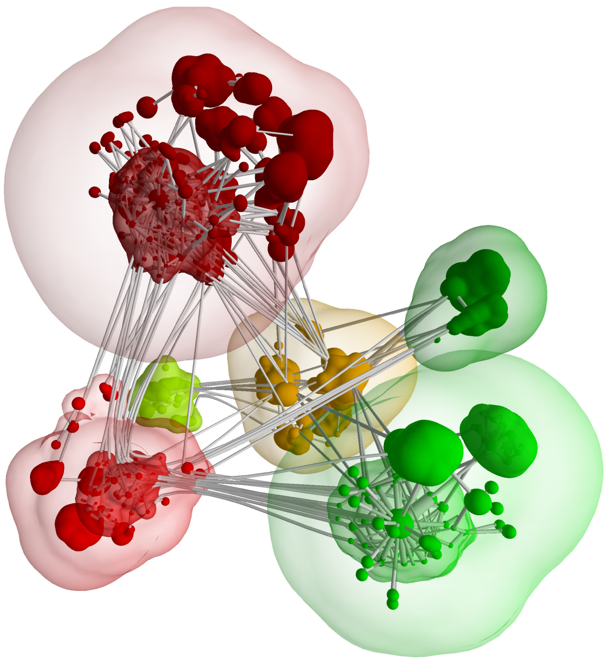


Figure 2.12: Clustered graph layouts.

The nodes in a given cluster are used to construct a scalar potential field sampled on a grid (regular or not) defined on the extent of the graph layout. The potential is high close to the nodes and low further away. By contouring the potential field at suitable values, using *e.g.* an isosurface constructing technique such as marching cubes or similar [57], smooth surfaces are created which are guaranteed to enclose all nodes. The technique is repeated for each cluster and the resulting isosurfaces are rendered using transparency to show intersections or nesting.

The implicit surface technique has been used in many visualizations that need to show visual containment of several elements in several clusters (which are conceptually equivalent to our areas of interest). Implicit surfaces have several advantages. First, they are easy to construct and reasonably fast to compute using modern isosurface extraction algorithms. Second, the visual nesting of elements in the implicit surfaces is guaranteed by construction.Third, the layout of the underlying graph that gives the positions of the elements is not constrained in any way. Fourth, the produced shapes are smooth, and have the look and feel of traditional Venn-Euler diagrams, which makes them easy to interpret. We shall see more examples of techniques using various forms of implicit surfaces in the following sections.

However, using implicit surfaces to show clusters or areas of interest has also several important disadvantages. First and foremost, the choice of the isosurface level is critical for the result. Isosurfaces are not connected by construction, so choosing a too high level (or too low if the potential field is low close to the elements and high far away from them) will create disconnected surfaces, which give the impression of several clusters, rather than a single one. Several heuristics and workarounds to this are presented in [91], but the problem is inherent to implicit surfaces. Second, it is hard to locally control the shape of the implicit surface. This shape is determined by the nature of the potential field used, and a simple construction of such fields will treat all nodes similarly, *e.g.* by summing up a radial potential function centered at each node, as described in Section 2.3.4 further on. This yields overall smooth surfaces, but makes it more difficult to create surfaces with a varying level of smoothness, such as rounded boxes, for example. Workarounds for this issue exist, but they require the construction of more complex potential functions, a more involved analysis of the spatial distribution of nodes, and higher computational times.

2.3.4 Graph splatting

Graph splatting [113] is a technique which transforms a graph into a two-dimensional continuous scalar field. Splat fields are useful to rapidly gain a overview of the complete structure of the graph. The scalar field can be rendered as a color coded map, a height field, or a set of contours. Splat fields allow for the visualization of arbitrarily large graphs without cluttering.

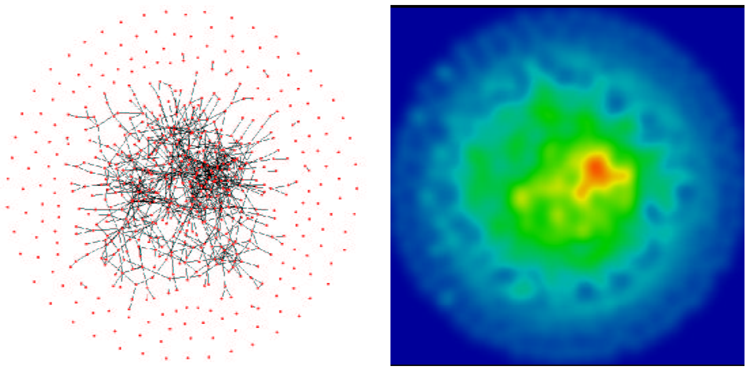


Figure 2.13: An example of the graph and its color coded splat field.

Graph drawing algorithms consist of two steps: layout and rendering. The layout phase constructs the desired layout, subject to the preferences of the user. In the original paper a spring embedder is used, but different algorithms can be used. The rendering phase, which is the phase we are concerned with here, transforms the graph in a continuous field F by summing up, or superimposing, a set of radial Gaussian basis functions or *splats* centered at the locations of the nodes

$$F(p) = \sum_{i=1}^{N} k e^{-f(p_i - p)^2} \tag{2.1}$$

where p_i are the positions of all graph nodes $1 \ldots N$, k are the weights (or importances) of the nodes, and f is a scaling factor controlling the splat size. By choosing different weights for different nodes, the constructed potential field can be made to emphasize stronger, or weaker, given nodes. For example, nodes with many edges, or having high values for some metric of interest, can be made more prominent in the visualization. The same idea can be used to splat the edges of the graph, by placing splats at several sample points along each edge.

Figure 2.13 left shows a graph displayed with a traditional node-link method and Figure 2.13 right shows the color coded splat field of the same graph. A rainbow colormap is used to show the splat field values: blue denotes low densities, while red denotes higher densities. The GraphSplatting technique allows to zoom into a region of interest. The color coding is adjusted to show values in the region with more details. Figure 2.14 illustrates splat field zooming. The left image shows an overview of the splat field with a region of interest. The right image shows the region of interest using zooming and a color adjustment based on rescaling to the splat values in the region of interest.

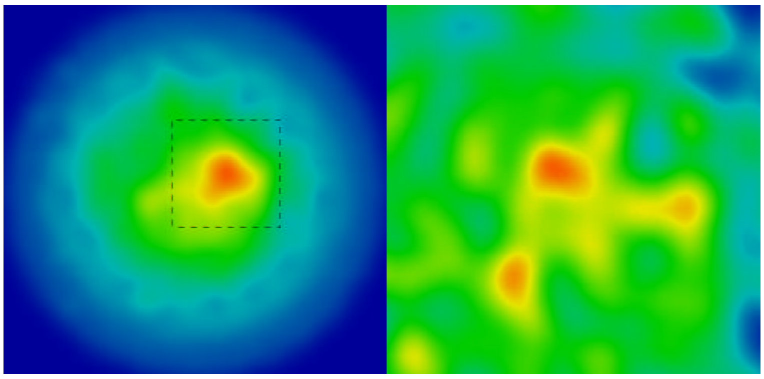


Figure 2.14: An example of zooming into an region of interest in a splat field

The utility of splat fields is based on the assumption that the density of points is a meaningful characteristic of the graph. That is, graph splatting mainly focuses of highlighting clusters of graph nodes which, due to the chosen layout, are spatially dense. The technique can be combined with isolining along the lines described in Section 2.3.3 to explicitly show clusters of nodes. These are conceptually equivalent to our areas of interest. However, as for the other uses of implicit surfaces to draw such areas of interest, graph splatting results are strongly dependent on the underlying layout to place related nodes close to each other. If we do not have such a layout, splatting will not be able to show nodes far away from each other as being related, or in the same 'area of interest'. Also, color-coded graph splatting cannot show different overlapping areas of interest.

Apart from color coded images and isolines, graph splatting results can be also visualized as 3D height plots, and have also been used in software visualization [103].

2.3.5 Thematic software maps

Kuhn *et al.* [1] propose a consistent layout for software maps in which the position of a software artifact reflects its vocabulary, and distance corresponds to similarity of vocabulary. They use *Latent Semantic Indexing* (LSI) to map software artifacts to a vector space, and then use *Multidimensional Scaling* (MDS) to map this vector space down to two dimensions. Conceptually, the layout technique is similar to force-directed spring embedders, but the implementation is different.

Consistent layout for software should make it easier to compare visualizations of different kinds of information, but software artifacts have no natural layout since they have no physical location. Kuhn *et al.* propose to use *vocabulary* as the most natural analogue of physical position for software artifacts, and to map these positions to a two-dimensional space as a way to achieve consistent layout, where distance between software artifacts corresponds to distance in their vocabulary. Additionally, they use digital elevation, hill-shading and contour lines to generate a landscape representing the frequency of topics.

The proposed technique was implemented in the Software Cartographer tool. Figure 2.15 shows the evolution of a software system using a thematic map approach. The changes in the same software system are shown from left to right. As all four views use the same layout, users can build up a mental model of the system's spatial structure and its changes.

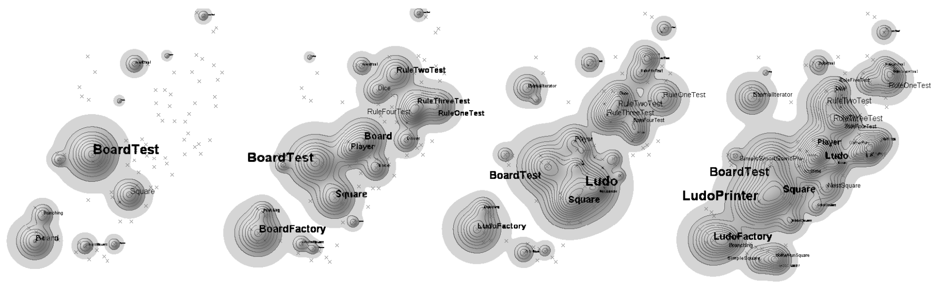


Figure 2.15: An example of thematic software maps.

The rendering of groups of related elements, or areas of interest, is done using implicit functions, in a very similar manner to the approaches presented in Sections 2.3.3 and 2.3.4. The same main limitations of implicit surfaces discussed before apply here: connectivity of the areas of interest cannot be guaranteed, and it is difficult to show intersecting areas of interest.

2.3.6 Enridged contour maps

The enridged contour map technique [42] was designed to show a sequence of (nested) contours, or isolines, of a scalar field. One added value of enridged contour maps consists in adding shading to emphasize the visual nesting of contours. Shading is dark close to a contour line and gradually increases to bright towards the following contour line, using *e.g.* a parabolic luminance variation profile. The user perceives the variations in shading as nesting of the contours, and can thus see which contours correspond to higher or lower values without additional graphical information.

Figure 2.16 shows a result of applying the enridged contour map technique. The technique works similarly to the implicit surface construction described earlier, in the sense that a scalar field is constructed from the original input scalar data, and then used for shading. In the example in Figure 2.16, the input scalar data is a superposition of some Gaussian functions centered at the locations indicated by the red dots, but it could be an arbitrary function. Several isolines of this function are shown using enridged contours. The shading and nesting of the contours conveys an idea of the data values - contours which appear at the top have higher values than the ones which appear at the bottom.

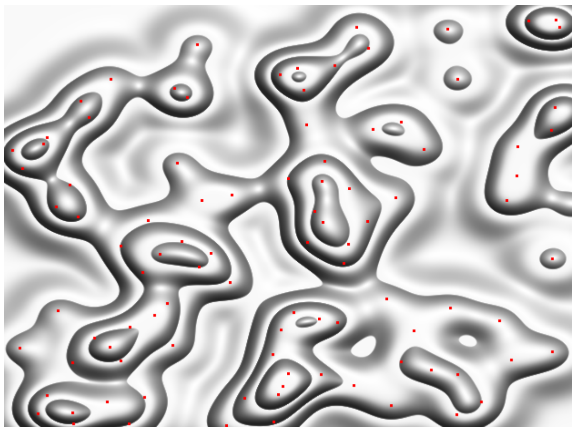


Figure 2.16: An example of enridged contour maps.

The enridged contour technique is interesting in our context from the perspective of shading. The smooth shading variations enhance the perception of nesting of contours. Similar uses of parabolic shading profiles to convey shapes exist in information visualization, most notably the cushion treemaps [115] and software structure cushions [56]. In the latter method, the syntactic structure of source code is emphasized by means of axis-aligned polygons whose borders are shaded similarly to the enridged cushion maps of [42] to convey nesting. In Chapter 6, we shall use a related shading technique to emphasize the extents of areas of interest in software architecture diagrams.

2.4 Conclusion

If we reflect back to our general goals stated in Chapter 1, *i.e.* the quest for a way to visualize software architectures using a UML-like look-and-feel, combined with metrics defined on diagram elements, element fields, and element groups, we see that there are several ways to improve upon the current state-of-the art in software visualization.

First, a considerable part of the software visualization metaphors in existence, with a focus on architectural data, use relatively abstract structural representation, like node-and-link graph layouts or nested layouts like treemaps. While it can be argued that this helps scalability, such views are also arguably less intuitive for software architects than UML diagrams. Moreover, diagrams are typically under hundred elements, so extreme scalability is not our central concern.

Second, only a few options for adding metric data to UML-like diagrams have been explored. Most solutions use scaling, coloring, texturing, or shading of the diagram elements (or of additional icons) to add metric data. This solution is not scalable in the number of metrics that can be shown at the same time on a given diagram element. Moreover, it is sometimes hard to compare different metrics or different values of the same metrics across a large diagram.

Finally, there is room of improvement related to the visualization of areas of interest, or groups of elements. There are only a few methods that render such areas in the style of Venn-Euler diagrams, which is arguably one of the most intuitive ways to do it. Problems concerning area overlaps and understandability need additional study.

In the following chapters, we shall attempt to address these problems. Chapter 3 discusses the rendering of areas of interest on UML-like diagrams. Chapter 4 presents a user study aimed at evaluating our area-of-interest rendering method. Chapter 5 shows how to render several method-level metrics having many values atop of the elements in a diagram. Chapter 6 shows how to render element-level metrics on diagrams such that these can be easily correlated with areas of interest.

Chapter 3

Areas of Interest

In this chapter, we introduce the Areas of Interest (AOI), a new element of system design information created with the purpose of emphasizing groups of diagram elements that share common design properties. We present an overview of the main requirements for the areas of interest and explain the reasons under which the design decisions were made. Next, we describe and compare two different methods for visualizing AOIs, and discuss their trade-offs from an algorithmic perspective. We present the application of AOIs on UML class and component diagrams.

3.1 Introduction

An important part of understanding complex software systems requires getting insight in how system properties, such as performance, trust, reliability, or code-level attributes, such as complexity or cohesion, correspond to the system architecture. Such properties can be seen as defining several *areas of interest* over the system architecture. Informally put, an area of interest, or AOI, consists of as a set of system architecture elements that share some common property of interest to the one analyzing the system.

In this chapter, we address one central question regarding areas of interest: How can we visually represent several, possibly overlapping, areas of interest on a given software architecture diagram, so that their graphical representation efficiently and effectively conveys the sets of elements sharing the underlying properties which the areas are built to show? We proceed to answer this question in a design-oriented fashion, the steps being as follows.

First, we give a more rigorous definition of areas of interest in terms of sets and data attribute values (Section 3.2). This also enables us to express the goals of an AOI drawing in a precise manner, and distill a set of requirements that such a drawing should comply with. As we shall see, these requirements are of various natures, the main ones being centered around visual understandability, scalability, and rendering speed (Section 3.3). These requirements are in line with the overall requirements for the visualization of design artifacts outlined for this thesis, as explained in Chapter 1.

Next, we present a first method for visualizing the AOIs - the so-called *inner skeleton* method. The inner skeleton method satisfies well the scalability and speed requirements, but has limitations in visual understandability (Section 3.4). We use the observations developed during the design and application of this method to create a second visualization method for AOIs - the *outer skeleton* method. As the naming suggests, the outer skeleton method uses geometric information defined on the outside hull of the elements in a given AOI, while the inner skeleton uses geometric information located inside this hull. The outer skeleton increases visual understandability by imitating the way humans would draw AOIs with pen on paper (Section 3.5). We discuss several extensions that further improve the understandability and scalability of the outer skeleton method in Sections 3.6 and 3.7. Sections 3.8 and 3.9 conclude our presentation of the AOI visualization with a general discussion and several examples on real-world class and component UML diagrams.

3.2 Data model

First, we introduce our data model for the areas of interest, in line with the software design representation presented in Section 1.3. As input information, we consider a system model (*e.g.* UML model), which contains a set of diagrams, such as class, component, or sequence diagrams, related to it. Formally, we define an *area of interest* (AOI) as a set of model elements. A model element m can be present in different diagrams D_i, in which case m will be shown by the corresponding diagram elements $d_i \in D_i$. Hence, in different diagrams D_i, the same AOI can be visualized as different sets of diagram elements d_i. The simplified data model of the areas of interest is shown in Fig. 3.1 using a UML class diagram.

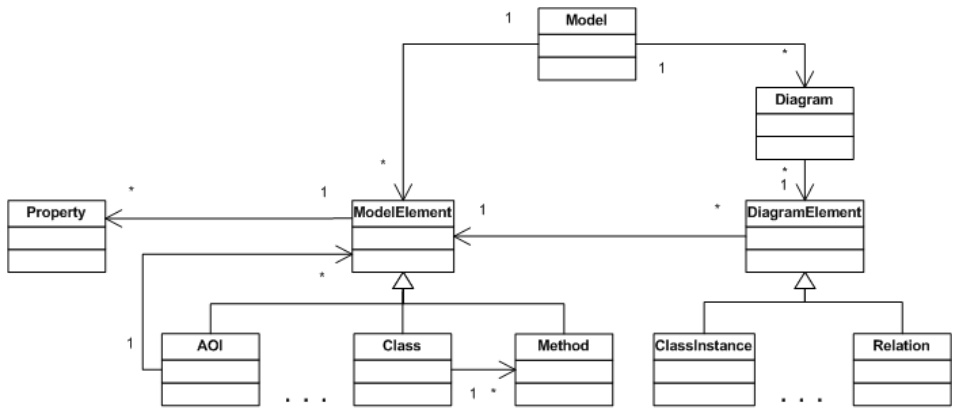


Figure 3.1: Data model

Usually, model elements are grouped in a given AOI precisely to reflect the fact that they share a common property of particular interest in a given system analysis. Simple examples of areas of interest built along this idea are: ”all high-reliability components”,

”all components using over 1 MB of memory”, ”all components introduced in the system version 2.3”, or ”all components in the same thread” [116].

As we see in the above examples, the common *properties* that are shared by the model elements contained in a given AOI essentially reflect the values of the data attributes of those elements. Hence, we distinguish two main ways to create areas of interest, based on the origins of the data attributes of the system elements:

- *manual construction:* AOIs, and their corresponding data attributes, are manually assigned and created by users. This can happen either in the process of iterative system design, but also in activities such as refactoring or reverse-engineering.
- *automatic construction:* AOIs, and their corresponding data attributes, are created in a (semi)automatic manner by system analysis tools. Data attributes take their values from various software metrics [28, 33] which can be computed by existing analysis tools [123]. Clearly, a wide range of AOI types and scenarios is possible in this case.

The AOI creation classification presented above is important as it helps us discover several characteristics of AOIs from the way these emerge in practice. Manual AOI creation, being essentially a human design activity, is strongly visual: In most cases, users effectively *draw* the desired AOIs around elements of an existing diagram. For example, Figure 3.2 shows an actual photograph of a design tool after a design session. The smooth-shaped, colored, contours indicate two areas of interest, whose intersection contains exactly one diagram element. In this case, the AOI definition and its visual representation are one and the same thing. This is an intuitive representation of AOIs, but it has the disadvantage that it needs manual work to be created. In contrast, automatic AOIs are not typically drawn, but represented implicitly using tables whose rows contain the diagram elements and columns the various metrics computed on the system. However easy to automate, a tabular AOI representation is quite unintuitive, as compared to the visual representation discussed formerly. For instance, a visual AOI representation can easily show which elements are in two AOIs at the same time; this is much harder to do when using a tabular representation.

Our goal is to bridge the gap between the AOI definition and the AOI visual representation sketched above. We want to enable users to define their AOIs using metrics and data attributes, and be able to automatically create visual representations, or visualizations, which resemble the ones which are drawn by typical users during design sessions.

In the next session, we refine the above goals in a set of detailed requirements regarding the drawing of areas of interest in software design diagrams.

3.3 Requirements overview

Understanding and communication of the structure and quality attributes of the architecture is essential during development and maintenance activities for all stakeholders who participate in them. We identify the main stakeholders concerned with software structure and quality understanding and knowledge sharing as shown in Table 3.1, following a similar analysis done in [30]:

Stakeholders	Function/Roles
Designers	share knowledge, take design decisions
Developers	take technical decisions, implement a system
Maintainers	take technical decisions, maintain a system
Testers	test and analyze a system, share knowledge
Development managers	share knowledge

Table 3.1: Stakeholders and roles

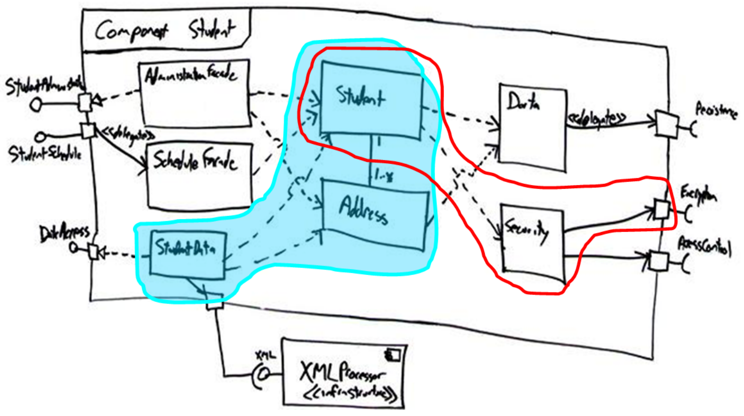


Figure 3.2: Whiteboard architecture drawing

From the above table, we see that sharing knowledge among a wide range of types of technical experts is a crucial ingredient of the involved activities. When considering AOIs as one of the datasets they work with, an immediate consequence is that AOIs should be represented (visualized) with a high level of *understandability*. This is our main requirement to the AOIs drawing. The way AOIs are drawn should be intuitive and easy to interpret for all stakeholders.

But what is the most understandable way to depict an AOI? Clearly, we cannot answer this question exhaustively, as this would imply trying out all possible visualization designs for an AOI. We address this question by returning to the manual creation of AOIs (Section 3.2). We assume, as a design start point, that an AOI visualization method which imitates the way humans draw AOIs with pen on paper should produce understandable results for a large class of users. To illustrate this, consider Figure 3.2 which shows an actual whiteboard-like drawing from a design session of a component diagram with two manually drawn AOIs (one drawn in filled light-blue and one drawn as a red outline).

The AOI visualization methods presented in the remainder of this chapter built around this assumption by trying to imitate, in an automatic fashion, several graphical elements that we identified when studying several human-drawn AOIs on system diagrams similar to the one shown in Figure 3.2. The elements which we identified as typical for a human AOI drawing are as follows

- AOI shapes are two-dimensional, just as the diagram drawings
- AOI shapes surround the elements which are logically contained in the respective areas
- AOI shapes are drawn without changing the layout of a given system diagram
- AOI shapes are typically soft, containing few sharp angles and straight lines
- AOI shapes have a 'sketchy' look, different from the precise look of the diagram drawing
- the pen style (thickness, sharpness) used to draw AOIs is different than the pen style used to draw the diagrams

A more detailed analysis of these design elements and the understandability of user-drawn AOIs is presented separately in Chapter 4.

Let us now detail the above observations. First, we limit ourselves to two-dimensional AOI visualizations. This is a natural conclusion of the fact that software system diagrams are predominantly drawn in 2D in practice, as reflected by the various UML design tools (*e.g.* [37], [106], [13]). Second, the fact that AOIs surround the contained elements is fundamental to the visual representation of the areas of interest, following the well-known model of Venn-Euler diagrams used in many fields, such as discrete mathematics. Third, the fact that AOI drawing should not change the layout of a *given* diagram follows naturally from the usage scenarios: In most cases, users want to represent concerns expressed by AOIs on a given, familiar system diagram. It follows that the diagram's layout, that is the positions and sizes of the diagram elements, should not be changed when adding AOI information. We do not want to change a given layout to show areas of interest, as this can destroy the user's 'mental map' and severely reduce understandability, a well known fact in information visualization (see *e.g.* [89]). Finally, drawing AOIs in a different graphical style (soft curves, using a fuzzy pen style) than the diagram (hard lines, sharp pen style) visually separates the two types of information, *i.e.* structure (diagrams) from attributes (areas), and also reflects that AOIs are added in a separate design process, *atop* a diagram and *after* a diagram is created.

We should note that our AOI drawing constraints, as inferred from our explicit desire to mimic human drawings, limits ourselves to a specific class of drawings. Several other ways of visually representing areas of interest, *i.e.* groups of related diagram elements, exist. Many software visualization tools such as Rigi [110], Prefuse [70], or MetricView [107], often used in reverse engineering and reengineering activities, make different representation choices. An overview hereof is given in Chapter 2. One possibility is to simply draw AOIs as rectangular boxes instead of our proposed soft curves. Drawing AOIs as boxes without changing the base diagram layout yields unacceptably high visual clutter and diminished understandability on general diagrams where the elements of an area can be scattered across the entire diagram. A second option is to change the layout of the diagram to bring elements logically contained in the same area(s) close to each other. While this solution can lead good results for a small number of area overlaps, this method destroys the user's mental map which is reflected by the given diagram layout. Also, this solution does not work when we have several AOIs which we want to show in sequence, rather than simultaneously, on a given diagram. It is unacceptable to change the diagram layout every time we want to show (or hide) a given area of interest. Finally, we mention the solution of visualizing an AOI by marking its elements with icons scaled, colored, and shaped to show metric values. Yet, inferring AOIs from such markers is hard for diagrams with many overlapping AOIs, as we shall see later.

Moreover, we would like to modify or explore our visualizations in real-time, even for large diagrams and many AOIs. For example, users should be able to quickly switch on or off a given AOI, change its drawing style, or even interactively re-layout the diagram.

We summarize now the requirements of our AOI visualization in Table 3.2. The requirements are numbered for easy reference in the following discussion.

Requirement	Description
UML-related	
M1	- AOIs must preserve the given UML diagram layout and drawing
Understandability	
U1	- the drawing technique should mimic the way humans draw the areas themselves
U2	- AOIs should be drawn with minimal visual clutter, even when they overlap
U3	- AOIs and other diagram elements should not visually interfere
Scalability	
S1	- AOIs drawing should be real-time, even for large diagrams and many areas

Table 3.2: Requirements to AOIs drawing

We next present two different methods for visualizing areas of interest, which attempt to comply with the above listed requirements. As we shall see, these methods have a number of particular advantages and drawbacks, and are therefore suitable for specific scenarios.

3.4 Inner skeleton solution

Our first method consists of two steps. First, we build a so-called *skeleton* of the AOI using the elements' geometric layout data, thereby addressing requirement (M1). We call this the *inner skeleton* of an area of interest, to distinguish it from the outer skeleton, which will be the subject of our second AOI visualization technique (see Section 3.5). Next, we draw the AOI using a graphics technique called texture splatting. By controlling the various splatting parameters, we shall address requirements (U1, U2, U3).

3.4.1 Inner skeleton construction

The input of the first step is the set of elements in a given AOI. For every element, we assume we have its geometric layout information in the diagram, *i.e.* the position and size of its 2D rectangular bounding box. We now build the skeleton of the AOI as follows (see also Figure 3.3, which illustrates the complete process on a simple AOI that contains three elements).

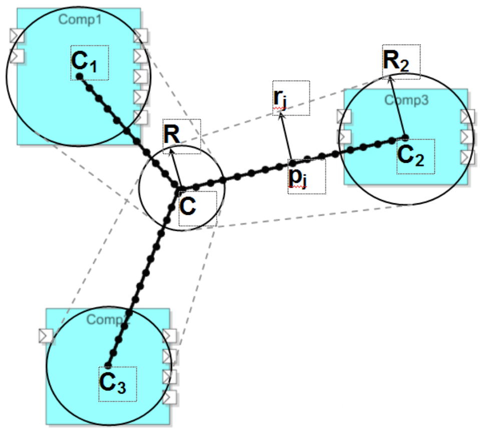


Figure 3.3: Geometric inner skeleton construction

For a diagram with elements e_i having geometric centers c_i, the skeleton is the line set (c_i, C), where

$$C = \sum_i A_i c_i / \sum_i A_i \tag{3.1}$$

is the area-weighted barycenter of the elements (A_i is the area of element c_i).

Given element e_i, with bounding box of width w_i and height h_i, a radius $R_i = \max(w_i, h_i)$ is computed for e_i and a radius

$$R = k_R \sum_i A_i c_i / \sum_i A_i \tag{3.2}$$

for the center C as a fraction k_R of the average radius. Setting the value for k_R is explained in the next section. Next, every line segment (c_i, C) is sampled with several points p_{ij} spaced with some small distance $\delta = |p_i - p_{i+1}|$, e.g. $\delta = 0.1R$. For every p_{ij}, we compute also a radius r_{ij} by linear interpolation between the radii R and R_i at the end of the segment (c_i, C). The final representation of the inner skeleton is the set of points and radius values $\{(p_{ij}, r_{ij})\}$.

3.4.2 Texture splatting

We now use the skeleton to draw the AOI, as follows. First, we construct a so-called *splat*. This is a radial function $T(x,y) = f(\sqrt{x^2+y^2})$. T looks as shown by Fig. 3.4 a (dark=opaque, light=transparent). Here, f is called the splat *profile*, or shape. We shall use $f(x) = x^k$, so T increases linearly with the distance for $k = 1$, exponentially for $k > 1$ and logarithmically for $k < 1$ (see Fig. 3.4 b). We implement T as a transparency (also called alpha) texture with the OpenGL graphics library [122]. Hence, $T = 0$ yields fully transparent pixels and $T = 1$ fully opaque pixels.

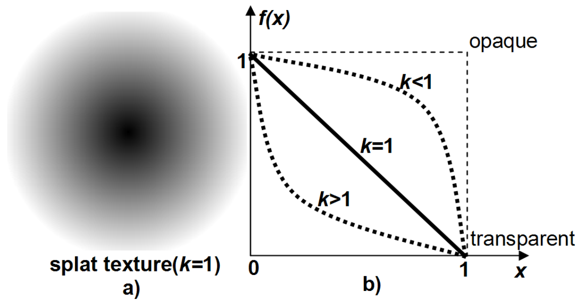


Figure 3.4: Splat texture(a) and texture profile (b)

The inner skeleton method renders the AOI by drawing the texture T centered at every skeleton point p_{ij}, scaled by the radius r_{ij}, and colored by a user specified AOI color. Figure 3.5 shows the result of the texture splatting for the AOI of the elements in Fig. 3.3. The texture splatting method presented above is conceptually similar with another tech-

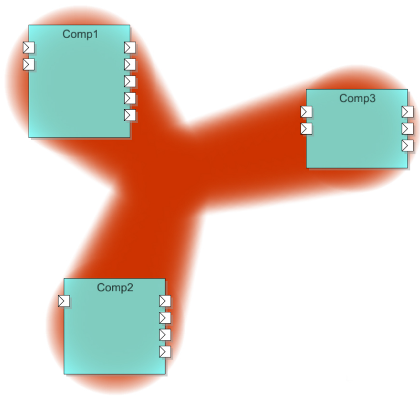


Figure 3.5: Area of interest drawn with splatting

nique, graph splatting [113, 103]. In graph splatting, a general graph is represented as a continuous scalar field by convolving (summing up) a set of radial splats centered at the graph node positions. While similar in technique, the two splatting methods serve different purposes. In graph splatting, the idea is to replace an entire discrete graph drawing by a continuous, smooth-looking, image where non-uniform spatial node densities are reflected by different values of the splatted signal. In our case, the aim is to construct a fuzzy shape (the AOI) where the transparency is low at the shape's edge and high at the shape's center. Moreover, we use a uniform point sampling density along the skeleton branches, since we want to achieve a uniform shape transparency along these branches.

Several properties of this method are visible here. First, the AOI drawing is visually quite different (i.e, soft and round) from the diagram drawing (drawn with sharp, straight lines). This distinguishes the two visually. Splatting the inner AOI skeleton is a robust, simple and fast way to draw a shape that contains all elements in an AOI and has a simple, predictable star-shaped look. This also explains the use of the word 'skeleton' in naming this method: the set of lines on which the circles' centers lie is, indeed, nothing else than the geometric skeleton, also called the medial axis in computational geometry, of the contour produced by splatting [87]. The resemblance goes further: in a different context, circular texture splatting of an arbitrary 2D shape's contour have been used to compute the geometric skeleton of a shape [95]. Here, we do the opposite: we splat the geometric skeleton, which we construct in advance, to obtain the corresponding shape's contour.

We let users vary k_R (see Sec. 3.4.1) to control the tightness and smoothness of the AOI shape. Small ($k_R \in [0.1, 0.5]$) values yield the typical tight star-shaped AOIs shown in Fig. 3.6 a. Large ($k_R \in [1, 3]$) values yield rounded, softer shapes (Fig. 3.6 d). In-between k_R values balance the trade-off between the shape smoothness and tightness (Fig. 3.6 b,c).

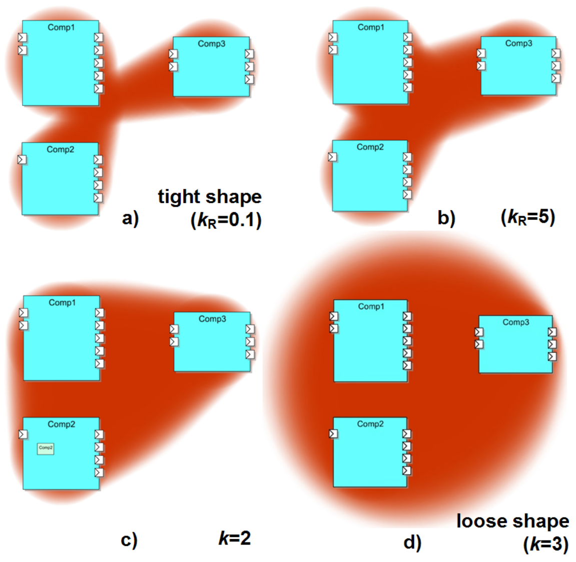


Figure 3.6: Area of interest drawn with inner-skeleton splatting

By controlling the various splatting parameters, we obtain visual effects useful for different user scenarios. If we want to draw 'hard' AOIs with a sharp, precise, border, we set $k < 1$ for the texture profile (e.g. $k = 0.3$, Fig. 3.7 a). This is useful *e.g.* to show important, prominent system properties or metrics having a high confidence value. If we want to draw 'soft', fuzzy AOIs, we set $k > 1$ (e.g. $k = 5$, Fig. 3.7 b). This is useful *e.g.* to show less important properties, which should not distract the eye from the more important diagram drawing, or metrics having a low confidence value.

A second variation our users found very intuitive and useful during our case studies (see Chapter. 7) was to draw AOIs as *contours* instead of filled shapes. For the inner skeleton solution, this is done in two passes. First, we draw the filled AOI using the splat textures, as described so far. Second, we draw the same AOI, using the same splat texture centered at the skeleton points, but now scaled to a smaller radius $d * r_{ij}$, and using the background color, e.g. white. $d \in [0, 1]$ controls the contour width: $d = 0$ yields the

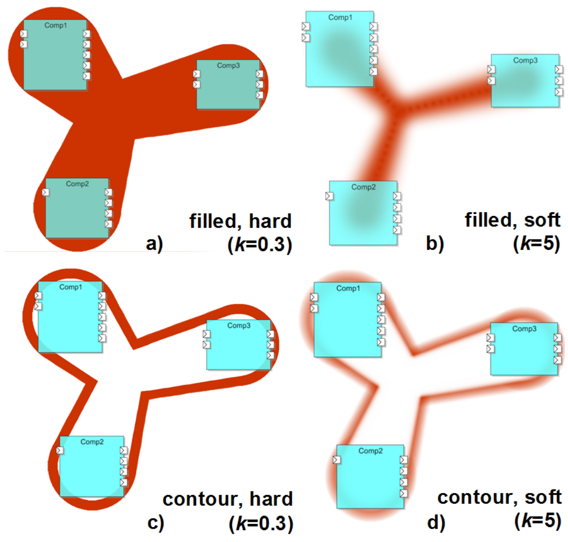


Figure 3.7: Filled and contoured areas

filled shapes, and $d \approx 1$ yields a very thin contour. As before, k controls the contour sharpness. Figure 3.7 (c,d) shows two examples of areas of interest drawn with contours with a contour width $d = 0.8$.

However, contour drawing using the inner skeletons has the unpleasant property that it erases the inside of the contour. This leads to undesired effects when e.g. drawing multiple, overlapping AOIs, as shown in Fig. 3.8. If desired, this problem can be eliminated by using multi-pass rendering techniques.

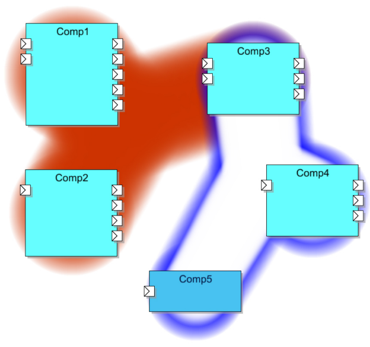


Figure 3.8: Inner contour-area overlap problem

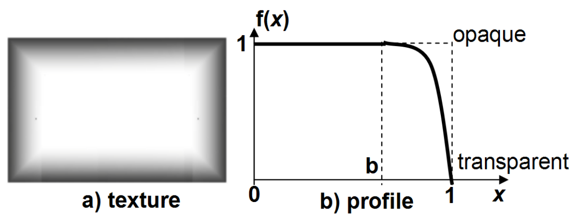


Figure 3.9: Eraser texture design

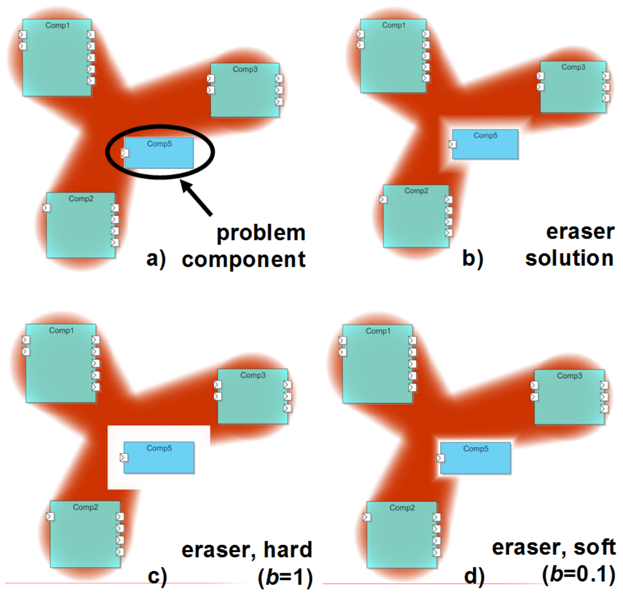


Figure 3.10: Erasing incorrectly overlapping elements

3.4.3 Exclusion problem

The inner skeleton drawing method, described so far, guarantees that the drawn shape visually surrounds all elements in the AOI. However, the drawn shape might surround, or overlap with, elements which are logically not in the AOI, but close to (or completely inside) that AOI, *e.g.* the element marked as "problem" in Fig. 3.9 a. This is, of course, an undesired side effect. One of our hard constraints is to never modify the diagram layout (see Table. 3.2 M1). Hence, we must find some other solution to visually show that such problem elements are actually not in the AOI they visually interfer with. We use the term *exclusion* to denote the process in which such elements are eliminated from the areas they inadvertently overlap.

We solve the exclusion problem as follows. First, we draw all AOIs as described so far. Next, for all elements not in any AOI, we draw an eraser texture. This is a transparency texture, like the splat texture (Fig. 3.4 a) used to draw the AOIs, except that it has a rectangular, instead of radial, shape (see Fig. 3.9 a) and a profile given by a slightly different function. Instead of $f(x) = x^k$, we use now the following profile f (see also Fig 3.9):

$$f(x) = \begin{cases} 1, & x < b \\ \left(\frac{x-b}{b}\right)^k, & x \geq b \end{cases} \tag{3.3}$$

Using a fixed $k = 4$ and varying b in $[0,1]$ yields an eraser ranging from hard $(b = 1)$ to very soft $(b = 0.1)$, as shown in Fig 3.10. The value $b = 0.8$ is a good default.

Drawing the eraser texture mapped on background-colored (white) rectangles slightly larger than the components effectively erases the AOIs underneath, yielding the effect shown in Fig. 3.10. The element that was erroneously overlapping with the AOI appears now to be outside the AOI. As for the splat textures, we can control the eraser strength by the k parameter, yielding results ranging from hard to soft (Fig. 3.10 c,d). A limitation of this technique is that it works only when combined with the filled style of AOI drawing, but not with the contour style. Another more effective possibility to avoid including such overlapping elements is discussed in Section 3.6

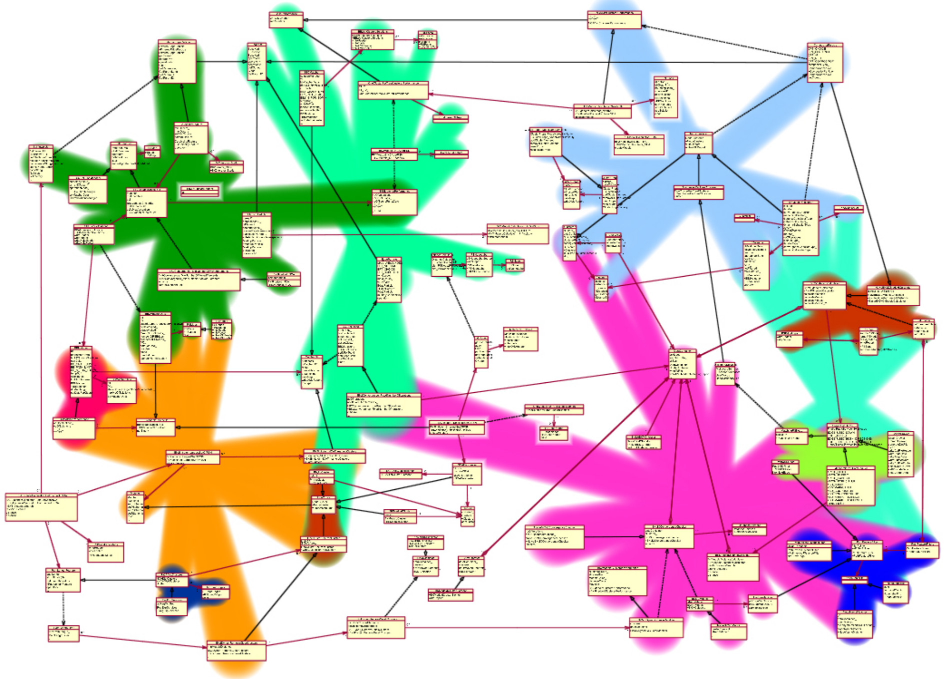


Figure 3.11: Area of interest drawn with splatting

3.4.4 Discussion

The main features of the inner skeleton method are its simplicity of implementation and predictable visual results. Implementing the complete method takes under 500 lines of

C++ code using OpenGL for the texture splatting. The resulting AOI shapes always exhibit a fixed, star-like, topology, *i.e.* a center point connected to the elements' centers. However useful, the inner skeleton AOI drawing has a major problem: It scales quite poorly for diagrams having overlapping AOIs of complex shapes, see *e.g.* Figure 3.11. This problem stems from the design limitation of the inner skeletons: they have a fixed, star-like, topology. Inner skeletons work quite well for small-size AOIs, containing under 5 elements, or AOIs whose elements' convex hull is geometrically close to a regular n-sided polygon.

However, the inner skeleton method produces less understandable results for more complex shapes, *e.g.* Figure 3.11. Moreover, the eraser technique described in the previous section, while effective in eliminating a small number of overlaps, can produce a distruptive effect when the number of overlapping elements is large and/or the amount of spatial overlap is large. In such cases, the rounded shape of the AOIs is abruptly interrupted by the rectangular cuts, yielding shapes which are hard to follow visually. Finally, the eraser technique does not work together with the contour drawing, as already mentioned.

All in all, the inner skeleton visualization technique for AOIs is successful in showing that AOIs can be effectively rendered atop of traditional system diagrams, using a different drawing style and no layout modification, albeit with a number of limitations. In the next section, we present a different AOI visualization method that keeps these desirable properties and also removes the limitations of the inner skeleton technique.

3.5 Outer skeleton solution

In our second methods, we create AOI visualizations also using a two-stage process. Firstly, a different kind of skeleton of the area, called the *outer skeleton*, is constructed using the geometries of the area elements. In the second step, areas are drawn using color and texture splatting. Finally, we propose a number of enhancements to the exclusion problem described in Section 3.4.3. These enhancements are detailed in Sections 3.6 and 3.7.

3.5.1 Outer skeleton construction

We first explain the outer skeleton construction. This process has three steps, see Figure 3.12b-d. We start with the 2D bounding boxes $(b_{1i}, b_{2i}, b_{3i}, b_{4i})$ of the elements e_i in the AOI (Fig. 3.12 a). We first compute the convex hull $C = \{q_i\}$ of the corner points $\{b_{ij}\}$, yielding the result in Fig. 3.12 b. This is the tightest convex polygon that encloses all our element bounding boxes, *i.e.* a possible approximation for an AOI shape. Still, we would like smoother, tighter fitting, shapes. To obtain this, we first subsample C (Fig. 3.12 c) such that the average distance δ between consecutive points $|q_i - q_{i+1}|$ is a given, small fraction of the convex hull perimeter $|C| = \sum_i |q_i - q_{i+1}|$. In practice, we set $\delta = 0.01|C|$.

Next, we deform the subsampled contour q_i so that it fits tighter the elements inside and, at the same time, yields a smoother curve than the convex hull (Fig. 3.12 d). We

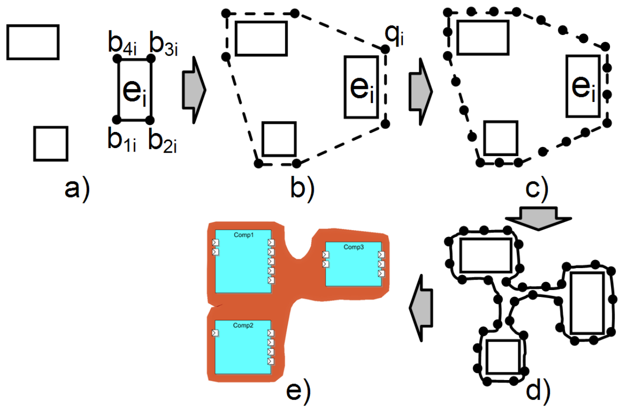


Figure 3.12: Construction of outer skeleton

deform the contour by moving every point q_i to q'_i:

$$q'_i = q_i + \varepsilon_n \vec{n} + \varepsilon_s \frac{q_{i-1} + q_{i+1}}{2} \tag{3.4}$$

Here, $\vec{n}$ is the normal to the line segment $(q_{i-1}q_{i+1})$. Assuming $\{q_i\}$ are specified in counterclockwise order, q_i will be moved inwards inside C. This serves two purposes. First, q_i moves perpendicular to the contour with a distance ε_n which shrinks the contour, making it tighter. Second, q_i moves towards the center of the line segment $(q_{i-1}q_{i+1})$ with distance ε_s. This is the well-known geometric Laplacian smoothing [99] with factor ε_s applied to our contour, which guarantees to remove contour sharp corners. We do the move in Equation 3.4 only if

$$d = \min(\min_{|j-i|>1} |q_i - q_j|, \min_j |q_i - p_j|) > 2\delta \tag{3.5}$$

i.e. the contour point q_i is farther from all element corners p_j and other contour points q_j (except its immediate neighbors q_{j-1}, q_{j+1}) than a distance 2δ. This test prevents the contour to self intersect during deformation. We move all points until we reach a user-set stop criterion or a maximum number of iterations N_{max}. Different stop criteria model different contour properties, as follows:

- Stopping when the deformed contour area $A(C)$ reaches a fraction $f_A < 1$ of the initial contour area controls the tightness of the AOI shape. Smaller f_A values mean tighter areas. Stopping after a given number of iterations $N < N_{max}$ does roughly the same and is also cheaper to implement.

- Stopping when the deformed contour length $|C|$ reaches a fraction $f_C > 1$ of the initial contour length controls the smoothness of the AOI shape. Larger f_C values mean less smooth contours.

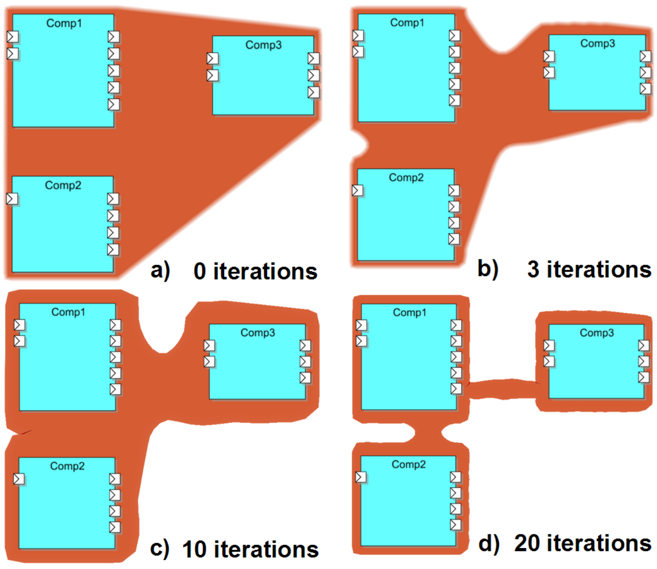


Figure 3.13: Controlling tightness in outer skeleton method

Figure 3.13 shows several deformation steps for a simple AOI, starting from the convex hull until a quite tight shape is reached after 20 iterations. The parameter setting $\varepsilon_n = 0.005|C| = 0.5\delta, \varepsilon_s = |q_{i-1} - q_{i+1}|/4$, $N \in [5..20]$ and $f_C \in [1,2]$ give very good results in practice for all configurations (shape, position, and number of diagram elements).

Besides preventing self-intersection, we must also prevent the contour to become too sparsely sampled, due to the contour length increase during deformation in concave regions. We do this by checking the distances $|q_i - q_{i+1}|$ and $|q_i - q_{i-1}|$ between the moved point q_i and its neighbors. If these exceed 2δ, we insert a new contour point halfway between q_i and the respective neighbor. Similarly, we check for the contour becoming too densely sampled, and remove sample points if they are at a distance smaller than $\delta/2$. Sample point removal occurs in convex regions of the contour which are moved inwards [17].

Fast convex hull and deformation computations are crucial for an efficient outer skeleton construction, given that we want near-real-time rendering. We use the Triangle geometric library [85] which provides a state-of-the-art convex hull implementation. For the deformation, the distance testing in Equation 3.5 must be done very efficiently. A naive implementation would use $O(NC(NC+E))$ operations per deformation step for NC contour points and E elements, which is too slow for real-time performance. We solve this by using a fast spatial search structure that locates the nearest point q_j to the moving point q_i in $O(log(NC+E))$ operations, using kd-trees [4]. All in all, these choices let us deform complex contours containing hundreds of elements (E) and hundreds of contour points (NC) in sub-second time.

3.5.2 Drawing the areas

We draw the AOIs using the outer skeleton in two steps. First, we triangulate the deformed contour $\{q_i\}$ (Sec. 3.5.1) and render the resulting triangles in the area's color. This takes care of the area itself. Next, we would like to draw a soft, fuzzy contour, similar to the effect in Fig. 3.6 for the inner skeleton drawing. We first tried the same idea of splatting the contour points with the radial texture. However, this requires a very high number of splats (roughly, one every few contour pixels) to produce relatively smooth border, which is quite inefficient. Using fewer splats yields a poor visual quality, where the individual splats are visible, see Fig. 3.14. The contour in Fig. 3.14 a is rendered with splats. We can see on the zoomed-in detail (Fig. 3.14 b) that, even though we are using a high splat density, the border looks jagged. We solved this problem by designing a better rendering

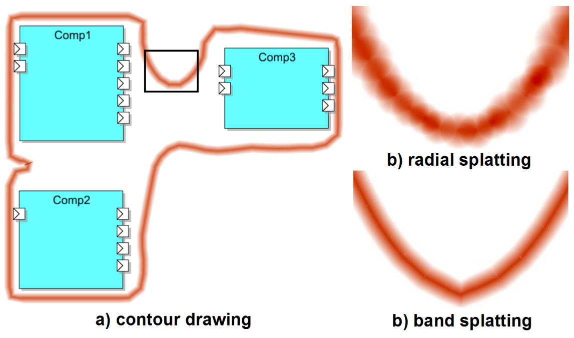


Figure 3.14: Contour splatting: (a) contour; details with (b) radial and (c) band splatting

method for the outer skeleton, as follows. We first offset the contour points q_i outwards along the contour normal $\vec{n}$:

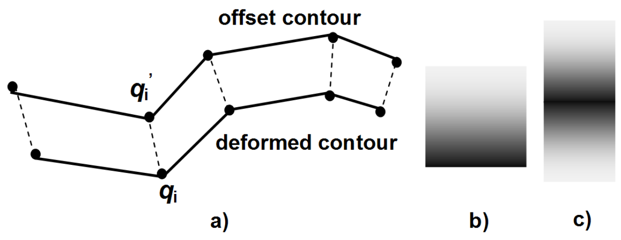


Figure 3.15: Soft border splatting for outer skeletons

$$q_i' = q_i + \varepsilon_n \vec{n} \tag{3.6}$$

Here, $\vec{n}$ and ε_n are the same as in Equation 3.4. This creates a narrow band along the contour (Fig. 3.15 a). Next, we create a 'band' texture $T(x,y) = f(x)$ (Fig. 3.15 b) where f is the same profile as for the splat texture (Fig. 3.4) and use it to render the border quadrilaterals $(q_i q_{i+1} q_{i+1}' q_i')$. This yields the soft border effect (Fig. 3.16) which looks very much like the soft edges of the inner skeleton rendering (Fig. 3.6). We can control the softness of the border by adjusting texture parameters, just as for the inner skeleton splatting described in Section 3.4.2.

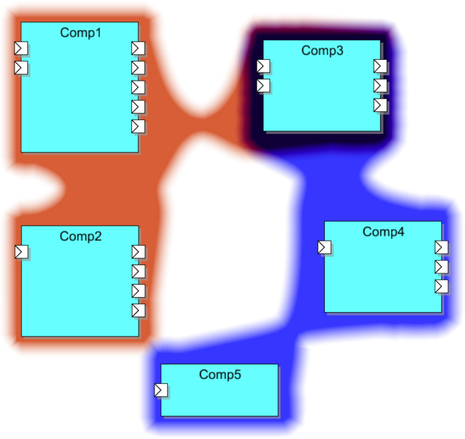


Figure 3.16: Outer skeleton contour

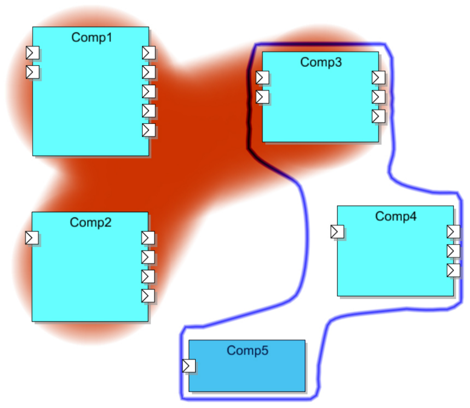


Figure 3.17: Two areas drawn with the outer skeleton technique

The problem of overlapping contours, discussed in Section 3.4.2 and illustrated in Figure 3.8, is easily solved when drawing AOI contours using the outer skeleton method (Fig. 3.17). The solution is to simply skip the drawing of the color-filled triangulation and to draw only the soft contour band, this time using a mirrored band texture (Fig. 3.15 c) to make the border look symmetric on the contour inside and outside.

As shown in Fig. 3.17, we can now easily understand which elements are in which AOI, *e.g.* the upper-right element is in both AOIs. After our users experimented with this display mode and some large diagrams (see Section 3.6 and Chapters 4 and 7), they required the same intuitive display of overlapping AOIs also in filled area mode, not only contour mode. To allow overlapping AOIs drawn in filled mode, we used a special blending mode, as follows. First, we render the background black. Next, we render all AOIs using $1 - RGB_i$, where RGB_i is the actual color of area i, in additive OpenGL blending mode. After all areas are rendered, we negate the image. The resulting color will be

$$RGB = 1 - (\max \sum_i (1 - RGB_i)) \tag{3.7}$$

The above can be interpreted as follows: Areas are rendered as before where they do not overlap. Overlap regions have a color equal to the subtractive blending of the overlapping areas' colors, *i.e.* overlapping regions show up as darker colors. The examples of color blending in overlapping areas is in (Fig. 3.25).

However, blending more than two colors, such as in the case of regions where three or more areas overlap, can create dark hues which are hard to distinguish. Another equally serious problem is that blending mixes colors, yielding hues which may not have any significance, or may even have a wrong significance, *e.g.* when the mix of two colors c_1 and c_2 of two areas A_1 and A_2 yields a color c_{mix} which is very similar to the color of some other unrelated area A_3. In Chapter 6, we shall present a solution to this problem using texture patterns in the different context of visualizing data values on areas of interest.

3.5.3 Mixing inner and outer skeleton AOIs

In comparison with the inner skeleton method presented in Section 3.4), the outer skeleton method reduces visual clutter, as the generated shapes follow the placement of the enclosed diagram elements more closely. The outer skeleton blending mode combines colors in regions where several areas of interest overlap in order to make such regions easily visible. The inner skeleton method still can be useful for small or middle-sized AOIs having a convex hull close to a regular n-sided polygon. For large AOIs or AOIs containing elements which are spatially scattered over large diagrams, the outer skeleton method produces results which are definitely easier to understand than the inner skeleton technique.

However, the two techniques are not mutually exclusive. Figure 3.18 shows the same large UML class diagram and areas of interest as in Figure 3.11, this time rendered with a mix of outer and inner skeleton techniques, as well as filled and contoured areas. For illustration, we have also included one area of interest drawn in a traditional style, that is, using a rectangular frame surrounding its elements (see Figure 3.18, lower-right). Clearly, the two rendering techniques and the filled and contour rendering flavors can nicely coexist in the same visualization. This variation in rendering techniques can be useful when emphasizing different system aspects, *e.g.* by using different rendering styles for different classes of properties.

3.6 Handling of exclusion

Albeit effective in rendering complex AOIs, the outer skeleton technique has an important limitation: it cannot directly handle elements which inadvertently overlap the extent of an

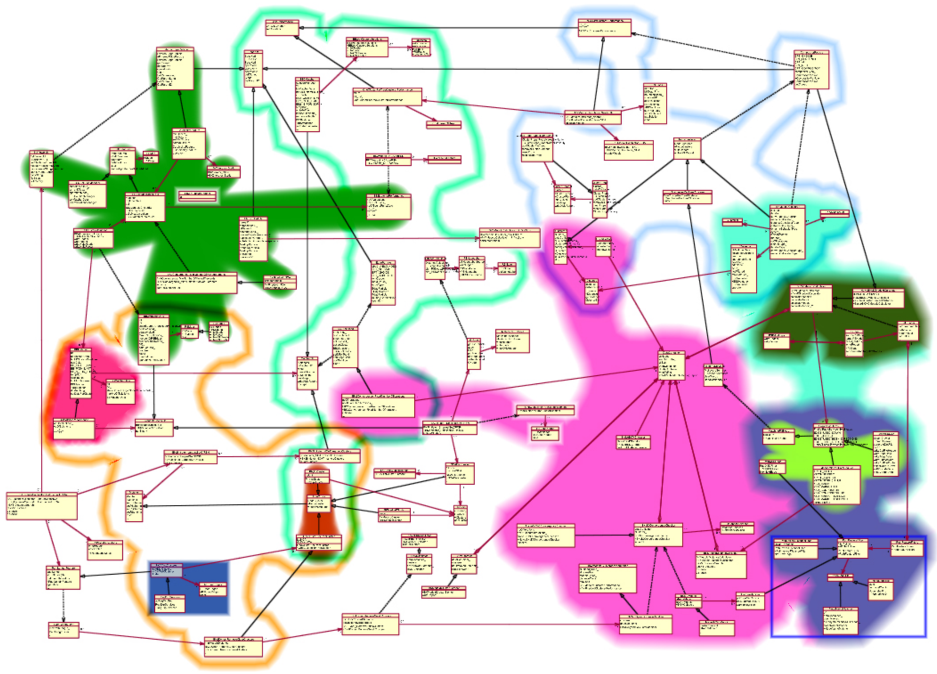


Figure 3.18: Mixed areas rendering using frames, inner skeletons and outer skeletons

area of interest, but are logically not inside that area - the so-caled *exclusion problem* introduced in Section 3.4.3. This problem is also exhibited by the inner skeleton method, as discussed earlier. We present next an effective solution to the exclusion problem implemented in the framework of the outer skeleton method.

3.6.1 Reviewing the exclusion problem

The contour constructed using the outer skeleton method described in Section 3.5 may erroneously overlap, or *include*, elements which are logically not in the AOI. In Section 3.4.3, we presented a method for marking such elements using an eraser texture for the inner skeleton method. The same method can be used for the outer skeleton technique. However, as we shall see below, a number of important limitations of the eraser texture method remain true for the outer skeleton case.

Consider Figure 3.19 where elements $A-D$ are in the area and E, F are outside. The eraser method works reasonably well if we draw *filled* areas and the overlapping elements are *completely* inside the area, *e.g.* E in Figure 3.19 a. However, even in this case the eraser cue is not salient enough to easily see that E is outside the area. For elements partially overlapping the area which need to be excluded, *e.g.* F in Figure 3.19 a, the cue is even weaker. For contour-drawn areas, the eraser technique works very weakly (F in Figure 3.19 b) or not at all (E in Figure 3.19 b) as there is little or nothing to

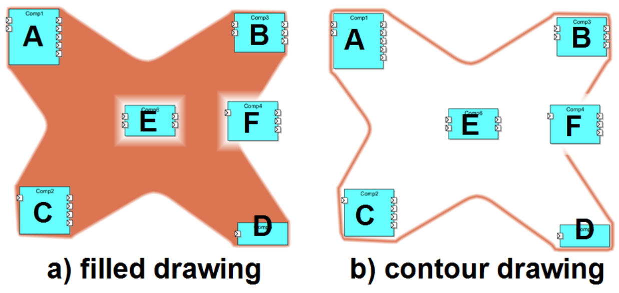


Figure 3.19: Limitations of the eraser technique. When drawing the AOI as a contour, element E is incorrectly shown as being inside.(a) Filled drawing and (b) contour drawing.

erase. This is a serious limitation since contour drawing is preferred to filled drawing in many situations, *e.g.* when one has only few available colors, when printing contours in black and white, when many contours overlap, or when blending-capable graphics hardware is not available. However, the most important problem of the eraser technique was that it turned out to be very unnatural for the most users who were shown it during our evaluations (Chapters 4 and 7).

In the following, we present an effective solution the exclusion problem in conjunction with the outer skeleton visualization method.

3.6.2 Geometric exclusion

Our main idea for tackling the exclusion problem is to edit the contour, before deforming it, in order to exclude the wrongly overlapping elements. The process works as follows (see also the scheme in Figure 3.20 and Listing 3.1).

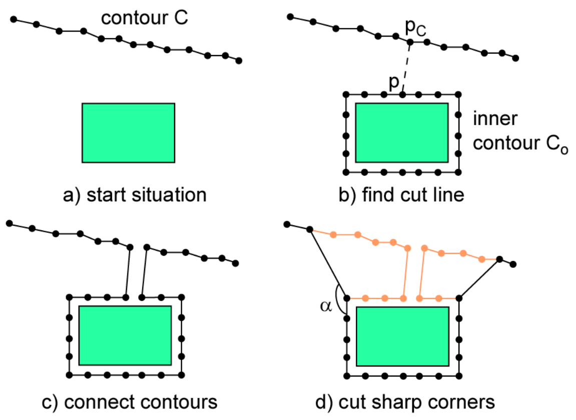


Figure 3.20: Geometric-based exclusion steps. (a) Start situation, (b) find cut line, (c) connect contours, and (d) cut sharp corners

```
compute overlapping element set $O = \{o_i\}$
```

```
for(all $o_i$ in $O$)
{
  create contour piece $C_o$ around $o_i$
  $p_o$ = point on $C_o$ closest to contour $C$
  $p_C$ = point on $C$ closest to $C_o$
  ready = false

  while (!ready)
  {
    //Move inner point left
    $p$ = $p_o$
    while($d(p, p_o) < d_{max}$)
    {
      if ($pp_C$ intersects no element $e_j$)
      { ready = true; break }
      move $p$ to left along $C_o$
    }
    if (ready) break;

    //Move inner point right
    $p$ = $p_o$
    while($d(p, p_o) < d_{max}$)
    {
      if ($pp_C$ intersects no element $e_j$)
      { ready = true; break }
      move $p$ to right along $C_o$
    }
    if (ready) break;

    //Inner move failed, do outer move
    move $p_C$ to left along $C$
  }

  connect $C_o$ to $C$ using line $pp_C$
  cut sharp corners
}
```

Listing 3.1: Iterative exclusion algorithm

First the overlapping element set $O = \{o_i\}$ is computed by testing, for all elements, if any of the four element corners falls within the already computed AOI convex hull C (Section 3.5). This requires a simple point-in-convex-polygon test which is fast and robust [20]. Next, each element $o_i \in O$ is excluded in turn, as follows. A finely sampled rectangular contour C_o is constructed around the bounding box of o_i (Fig. 3.20 b). Next, a short cut line connecting C_o with the original contour C is computed such that it does not intersect any of the elements ei in $e_i \in C$. To do this, we use the following heuristic. We find first the closest two points $p_o \in O$ and $p \in C$. Next, we move both p_o and p_C along the inner and outer contours O and C respectively, until the line does not intersect any element. We start by moving p_o around O to the left (counterclockwise sweep) until a non-intersecting line is found or a too high distance d_{max} from the starting position, as computed along O, is reached. If no line can be drawn, we try now moving p_o to the right (clockwise sweep). If this fails too, then we move the other point p one step along the outer contour C, and repeat the inner contour sweep again. When a cut line was found (dotted line in Figure 3.20 b), we connect the inner and outer contours by constructing two sampled line segments, close and parallel to the cut line (Figure 3.20 c).

The contour editing described above can create contours having unnecessary sharp corners. We reduce these in a separate pass. For each point p_i along the contour, we compute the angle $\alpha = \widehat{p_{i-1}p_ip_{i+1}}$ made by that point with its two neighbors. If the angle drops under a minimal value α_{min}, and the line $p_{i-1}p_{i+1}$ does not intersect any diagram element e_i, then we remove p_i from the contour by connecting p_{i-1} and p_{i+1}. Good values for α_{min} are in the range $[40, 70]$ degrees. We repeat this procedure iteratively until no removal is possible. The final result is shown in Fig. 3.20 d. When excluding several overlapping elements o_i, sharp corners are removed after excluding each element o_i, and not at the end. This gives better quality, as unnecessary sharp corners are eliminated as soon as possible. This also accelerates the further smoothing steps, since the contour gets simpler (that is, has less points). Finally, the shrinking process, which moves contour points in normal direction with constant speed, is more stable if sharp corners are eliminated, a well-known fact from the geometric level-set theory [84].

Figure 3.21 shows the same diagram as in Fig. 3.19. We see how the elements F and E are iteratively removed (Figures 3.21 b and c). The red line shows the contour after exclusion and sharp corner removal. The dotted black line shows the contour after exclusion but before sharp corner removal. Figures 3.21 d-f show the result after doing a few smoothing steps. Clearly, these results are better than the initial one in Figure 3.19 b. The exclusion algorithm now very clearly shows what is inside, and what outside, an area. The unnatural eraser effect is now gone. The sharp corner cutting procedure has the additional positive effect of smoothing the contour, yielding a more natural ’flow-of-hand’-like drawing.

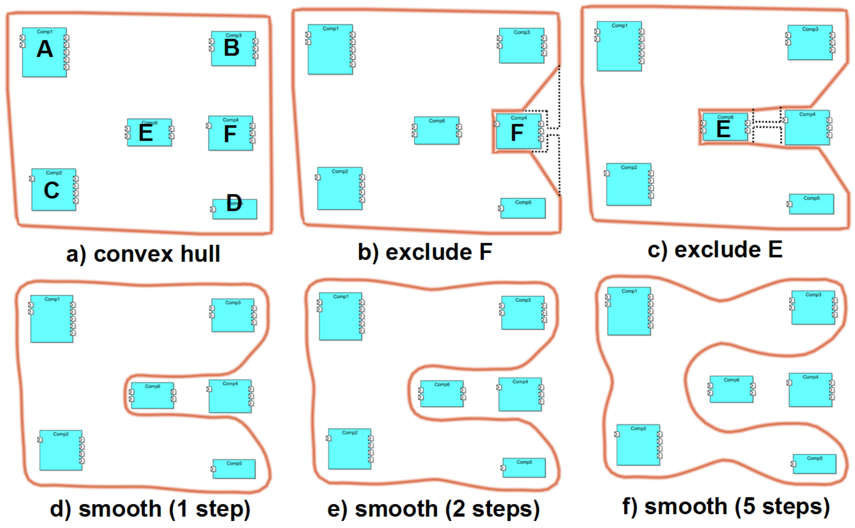


Figure 3.21: Geometric-based exclusion.

3.6.3 Limitations and workarounds for excluding elements

There are situations when the above heuristic for element exclusion (or any other algorithm, for that matter) cannot find a cut line that connects the element to be excluded witht he AOI contour without hitting some other element contained in that area. This occurs, *e.g.* when the element to be excluded is completely surrounded by a ring of elements which are in the area (see *e.g.* Figure 3.22). This situation can be easily detected algorithmically by monitoring when point p_C has executed a full loop over the area's contour C (line 32 in Listing. 3.1). Although such configurations would not be typically found in most software architecture diagrams, we discuss below two methods to handle them.

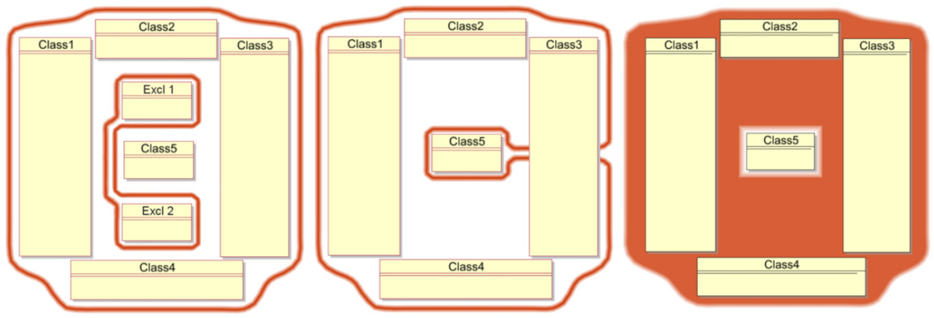


Figure 3.22: Exclusion algorithm for cases when no straight cut from the excluded element(5) to the area's contour is possible. Shortest-cut solution (left) and eraser solution (right)

The first solution groups all elements in O from an Area A which cannot be excluded using the cut technique in a new AOI A_{excl}. Since these elements are completely blocked from seeing A's contour, they must be fully contained in A. We now show the exclusion of these elements from A by drawing the contour of A_{excl} using the standard AOI algorithm, *i.e.* taking care to exclude elements which are in A but not in A_{excl}. Next, we draw the contour of A ignoring the exclusion. Figure 3.22 (left) shows an example. The Area A logically contains the elements 15 but not the elements $Excl1$ and $Excl2$. The latter two cannot be handled by the cut line technique, so they constitute A_{excl}. Hence, we draw the contourof A ignoring $Excl1$ and $Excl2$ and separately the contourof A_{excl}, this time taking care to avoid element 5, which is not in A_{excl}. The A_{excl} contour is nested in the A contour, since the elements in A_{excl} are completely within A. This technique handles well a large range of configurations. However, it would fail when the inner contour drawing (A_{excl}, which involves the cut line method, fails from precisely the same reason the initial exclusion of its elements from A failed. This happens in configurations involving several concentric rings of elements which alternately belong to A and A_{excl}. Although we could handle such situations by the addition of more contour pieces, this would create AOIs with several disconnected components which are increasingly hard to follow visually.

From discussions with actual software engineers who use UML in their daily work and tested our tool on repeated occasions, we observed a net preference for rendering AOIs as simply connected contours (shapes without holes) rather than multiple (inner and outer) contours. The main explanation given was that simply connected shapes are easier to follow visually, especially in complex diagrams with several areas whose contours overlap. In such cases, one needs to visually follow an area’s contour to discern that area, so an area with multiple disconnected boundaries may be wrongly perceived as several separate areas. Secondly, there are cases when one only has a few (or no) colors to draw the areas, *e.g.* in print-outs. In such cases, it is hardly possible to group disconnected contours as inner and outer boundaries of the same area, since color discrimination is not available.

A second solution for the cases when the cut line algorithm fails due to blocked contour visibility is to use as cut line the shortest segment linking the element(s) to exclude with the AOI’s contour, *i.e.* the segment (p_o, p_C) computed as in lines 5 and 6 in Listing. 3.1. Although this cut will intersect (at least) one of the elements which are logically inside the area, it has the desirable property of being the shortest possible one, thus the visually least disturbing, and also it generates a simply connected contour.

Figure 3.22 (middle) shows such a situation. The AOI contains elements $1 \ldots 4$ but must exclude element 5, which cannot be connected via a straight path with the area’s contour. Since no non-intersecting cut line can be found, the shortest cut line is used, which will intersect element 3. To further emphasize this cut, we skip the sharp corner cutting and smoothing (described in Section 3.6.2 and Chapter 4) for this cut. The cut will, hence, stay thin and have a visually distinct appearance from the regular cuts (compare Fig. 3.22, left, with Figure 3.21 f). For comparison purposes, Figure 3.22 (right) also shows the original eraser technique described in Sec. 3.4.3, which only works on filled areas. Overall, the preferences informally observed ranked the shortest-cut solution as the most accepted (Figure 3.22 middle), followed by the multiple contours (Figure 3.22 left) and finally the eraser technique (Figure 3.22 right).

3.7 Natural flow-of-hand

Figure 3.19 shows also a second problem of the original AOI rendering technique (apart from the exclusion problem discussed above). Close to the elements, the contours are too tight. In the middle, they are too loose. The contour smoothness is not optimal, as it looks too much like a sharp-angled polyline. The non-uniform tightness and sharp angles create a computer-made, unnatural contour look, quite different from the *flow-of-hand* typical to human drawings.

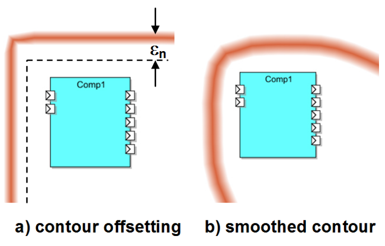


Figure 3.23: Contour smoothing close to elements

We achieve a more uniform tightness along the entire contour by the following preprocessing step. Right after the convex hull is sampled, and before the exclusion begins, we offset every point p_i on the contour C outwards with a small distance equal to the shrinking step ε_n. Figure 3.23 a shows this process on a zoom-in at one of the corners of the diagram in Fig. 3.19. The offset makes the initial contour looser, which gives it further space to deform and nicely curve itself around the elements (Fig. 3.23 b). The usage of this technique is also shown in Fig. 3.21.

The natural flow-of-hand technique presented here is easy to provide in the framework of the outer skeleton method, as this method explicitly models the contour of an AOI as a set of points. In contrast, such a technique would be not possible in the framework of the inner skeleton, which uses an implicit contour representation given by the linear variation of the circles' radii that get splatted (Section 3.4.1). By definition, the inner skeleton produces shapes that are symmetric with respect to the star-shaped skeleton, while a natural flow-of-hand contour requires more freedom in defining the shape.

3.8 Discussion

3.8.1 Improved AOI rendering method

Our improved method arguably brings the results of AOI drawing algorithm closer to the results of actual human drawings. To illustrate this, let us consider three drawings of areas of interest on the same diagram (Figure 3.24).

The top image is an actual scan of one of human drawings, done in a user evaluation described further in Chapter 4. The middle image shows the result of the our AOI drawing method produced on the same diagram using the eraser technique (Section 3.4.3). In this drawing, we recognize all problems named so far: Some elements (A,B,C) are incorrectly included in surrounding areas, whereas they should be outside, as shown in the top drawing; and the contours are tight and sharp close to the elements but loose and smooth in the middle. The bottom drawing shows our rendering method which uses the geometric exclusion (Section 3.6.2). The elements A,B,C are now correctly excluded. The contours have a more uniform smoothness and are not so tightly close to the elements. This drawing looks of a higher quality as compared to the one produced by the original algorithm. Probably the most appealing fact is that the areas drawn here simply *look* natural and quite similar to the human-drawn one shown in the top image in Figure 3.24. A more detailed analysis and measurements of similarities of computer and human-drawn AOI drawings will be presented separately in Chapter 4, in support of our claims of understandability.

Following the presentation so far in this chapter, we conclude that the outer skeleton method, complemented by the geometric exclusion and flow-of-hand techniques presented in the last two sections, is our method of choice for visualizing areas of interest from the point of view of understandability.

To illustrate the scalability aspect of this method, let us consider Figure 3.25, which shows an example featuring twelve areas on a UML class diagram with 110 classes, drawn as filled contours. The right zoom-ins show the areas framed in white on the left images. We see now the two problems of the method without geometric exclusion: Class *A* is incorrectly marked as contained in area 1. This is because the eraser (Section 3.4.3) is

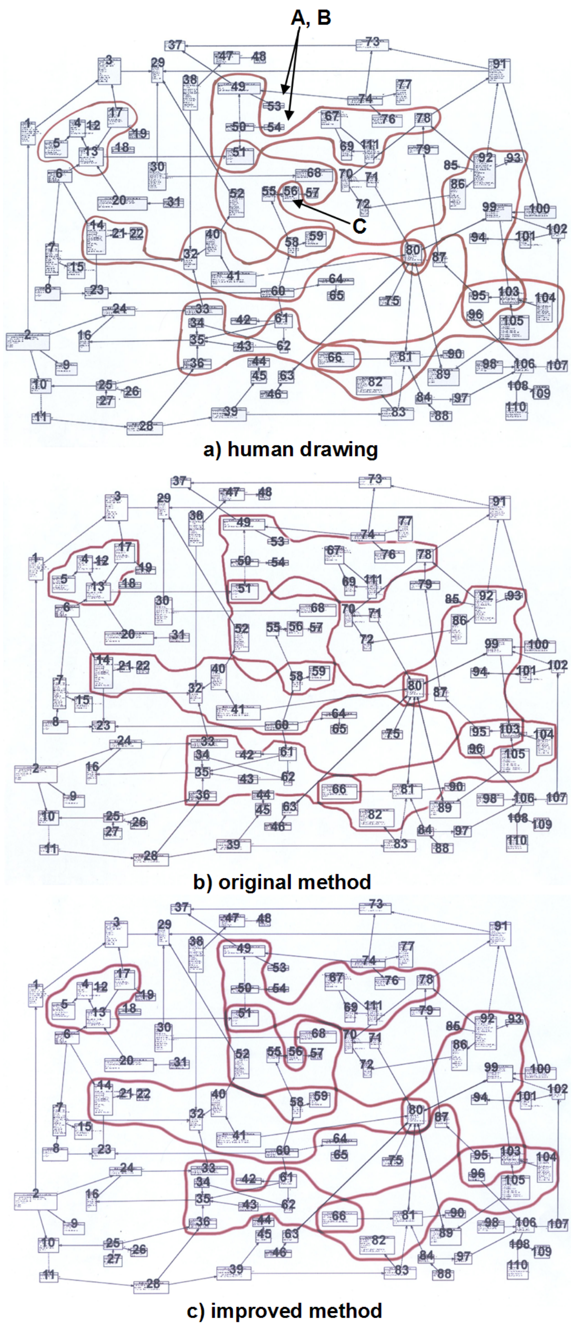


Figure 3.24: Comparison of AOI drawings (scans of the paper images)

overwritten by the color of area 2, in which *A* is indeed included. Class *B* is only in area 2, so the eraser is visible as a faint white border. However, when the diagram is zoomed out, this eraser becomes almost invisible. The improved method (Section 3.6) remediates both problems. Now the inclusion of *A* in Area 2 and the fact *B* is outside both Areas 1 and 2 is clear.

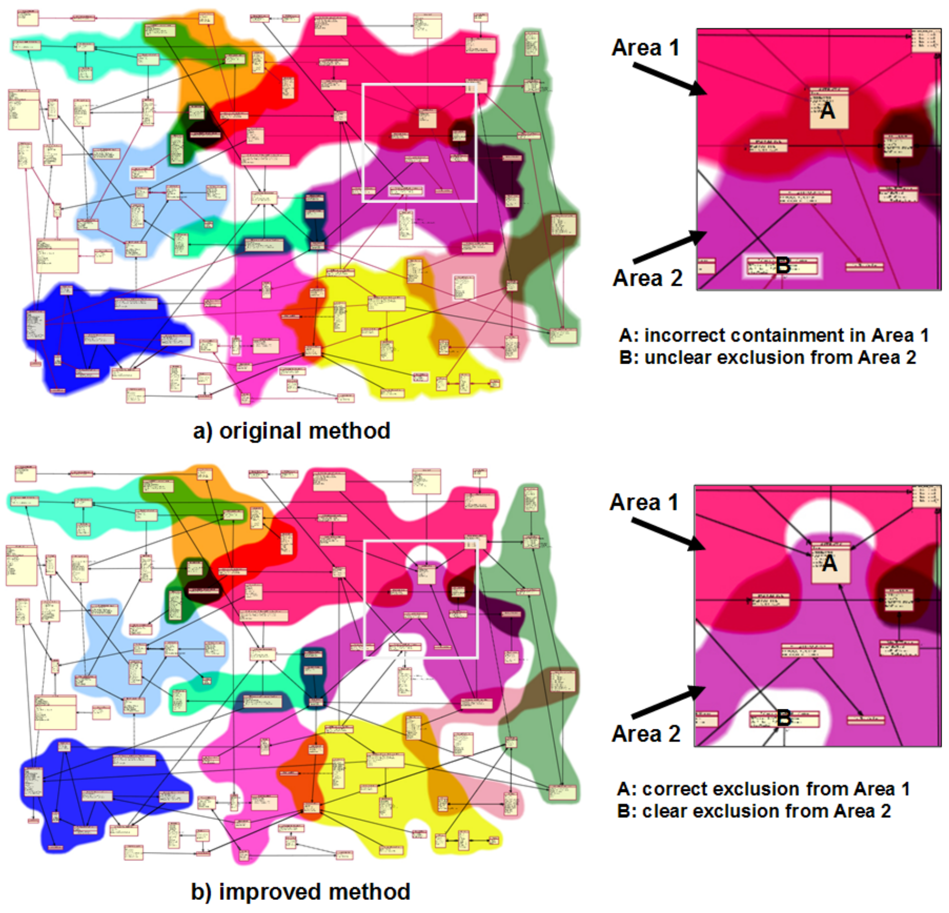

Area 1
A
Area 2
B
A: incorrect containment in Area 1
B: unclear exclusion from Area 2
a) original method
Area 1
A
Area 2
B
A: correct exclusion from Area 1
B: clear exclusion from Area 2
b) improved method

Figure 3.25: The improved exclusion method

The AOI rendering technique can be used on different types of diagrams besides class diagrams. Figure 3.26 shows two areas of interest rendered on a message sequence chart. Here, the new exclusion technique is crucial, due to the typical layout of such diagrams. Usage AOIs to show performance related-parameters of component-based systems is discussed in Chapter 7. Generally speaking, the AOI rendering technique can be used to emphasize logical subsets on many types of graph-like diagrams.

The improved method (Section 3.6) does not introduce new parameters that the user has to explicitly tune. The main user parameters, i.e. AOI color, drawing mode (filled

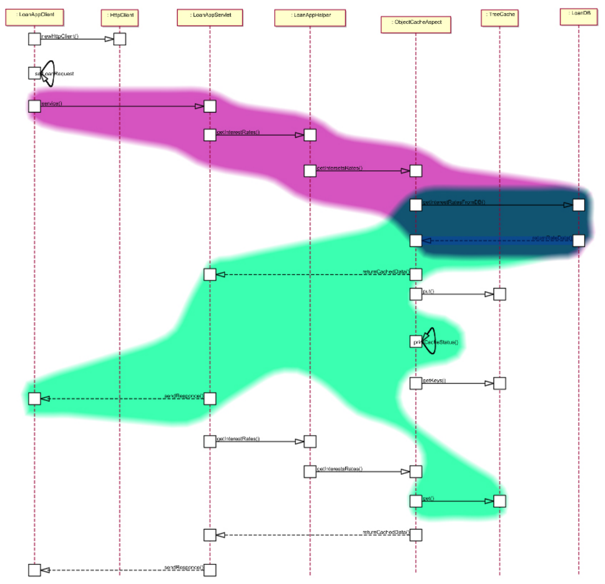


Figure 3.26: Example of drawing AOIs on message sequence diagrams

or contour), AOI transparency, and contour tightness, are still the same as in the original method. In particular, the relatively complex geometric exclusion algorithm is fully automatic. The speed of the improved method is slightly (about 10-15%) worse as compared with the original method. This is due mainly to the exclusion process which has to find an optimal cut line (Section 3.6) and also cut corners, both being iterative processes. However, we should stress that we have not highly optimized this code. Standard geometric optimizations such as spatial search techniques [4] can easily eliminate this performance decrease. Even with the performance drop, the AOI rendering still occurs in subsecond time.

3.8.2 Incorporating edges in areas of interest

In some cases, it may be desirable to constrain not only elements of a diagram to be contained in a given AOI, but also edges denoting relationships between such elements. For example, if two elements e_1 and e_2 are contained in an area A, then one may desire to constrain A to also include all edges (e_1, e_2) between them. A typical use-case hereof is emphasizing structural patterns such as design patterns. This can be achieved by adding sample points on the edges whose both end-elements are contained in a given AOI to the set of sample points generated on the elements' bounding boxes. The AOI construction algorithm will treat all these points uniformly, *i.e.* deforming the contour and constructing the cut lines without getting too close to them. When using straight-line segments to represent edges, these are contained by default in the initial convex hull that encloses all the area elements, so the only difference is the increased number of sites (points) taken into account during the area deformation and exclusion step. Note that precisely the same technique can be used for handling elements with non-rectangular shapes.

This technique works very well for diagrams having areas with little overlap. Consider, for example, a class hierarchy drawn on several layers as a tree or directed acyclic graph. The elements' convex hull will automatically include all edges in this hierarchy. The deformation (shrinking) step will move the contour inwards, as usual. If sample points on these edges are explicitly added to the AOI, the contour shrinking will stop when getting in their proximity, the rest of the algorithm staying unchanged.

However, when several areas considerably overlap and they also contain many edges, the blocking configuration described in Sec. 3.6.3 may occur more frequently. Such situations can generate overly complex curved contours, which, again, are hard to follow visually. An alternative solution to handle the logical edge inclusion in AOIs is to refrain from geometrically including them in the areas, but mark them with the color(s) of the area(s) they are part of. Though not ideal, this solution is robust and easy-to-implement.

3.9 Conclusion

The methods for rendering areas-of-interest presented in this chapter have been tested on a large set of tens of UML diagrams, both hand-drawn and automatically extracted from source code. In virtually all cases, the results produced by the improved rendering method (Section 3.8.1) were satisfactory, in the sense of being easy to understand and close to human-style drawings. We only noticed a few pathological cases when the outer-skeleton method was not able to construct suitable shapes, beyond the cases described in Section 3.6.3. However, these so-called pathological cases contained configurations such as several overlapping elements (due to an incorrect construction of the original input layout). We can safely assume that actual UML diagrams used in software engineering do not contain such configurations. Alternatively, an actual production-quality implementation of our AOI algorithm could easily detect such cases and refrain from attempting to construct AOIs in those situations.

In terms of understandability, the computer-drawn AOIs are close to human-drawn ones. As this is an important requirement of our research (see Chapter 1), we shall study the differences between the two types of AOIs in greater detail in the next chapter.

Chapter 4

Evaluation of Areas of Interest

In this chapter, we present a qualitative evaluation that delivered insight in how users perceive the quality of computer-drawn AOIs as compared to hand-drawn diagrams. Besides the user evaluation, we present a quantitative analysis to compare different AOI drawings, based on a distance metric defined between contours in image space. The results of this study are twofold. First, we obtained an empirical justification for the design criteria which are used in the computer-based construction of AOIs, based on what users perceive as important understandability features. Secondly, we obtained a quantitative assessment of the fact that our improved rendering method for AOIs, described earlier in Section 3.6, produces indeed results closer to good human drawings as compared to the initial visualization design presented in Section 3.5.1.

4.1 Introduction

Areas of interest, introduced in Chapter 3, are defined as groups of elements of system architecture diagrams that share some common property. Visualizing AOIs is a useful addition to plain diagrams, such as UML diagrams. In the previous chapter, two main methods have been presented to automatically draw AOIs on architecture diagrams: the inner skeleton and the outer skeleton method. The presented methods render AOIs as soft, fuzzy shapes surrounding the diagram elements, by a combination of geometric and texture-based techniques. The rendering method scales computationally well to tens of areas and hundreds of elements.

For the outer skeleton method, a number of improvements were presented that attempt to produce a contour which is close to the way humans would (like to) draw an AOI using pen and paper. Informal evaluations done during the design of the methods, as well as actual user tests done in a software engineering project (see Chapter 7) gave us the strong impression that the improved outer-skeleton method produces the best results, and moreover, results which are similar to human-drawn areas.

However, some important questions still remain to be answered: Do actual users like computer-drawn AOIs comparably to hand-drawn AOIs? If not, why, and how can we improve the computer-drawn AOIs so that they resemble more closely good-quality hand-drawn ones? In particular, are the improvements to the outer skeleton method discussed in Chapter 3 indeed useful? If yes, which are the actual features of the drawing which are most important for acceptance by users? Understanding these aspects is important both in validating the choices made during the AOI design process, and also for further work in improving the AOI drawing quality.

To evaluate the quality of AOIs, we designed and executed a detailed empirical evaluation. This evaluation is presented in this chapter. From the evaluation results, we distilled salient strengths and weaknesses of the original AOI rendering algorithm (Section 3.5) and of hand-drawn areas. Our conclusion was that hand-drawn areas, although quite variable across different humans, are perceived as easier to understand than computer-drawn ones. Two main drawbacks of computer-drawn areas were found: eraser-based exclusion of overlapping elements (Section 3.6.1), and unnatural flow-of-hand (Section 3.7). These are precisely the drawbacks that out improved outer skeleton method attempts to correct. After introducing the geometric exclusion (Section 3.6.2) and natural flow-of-hand (Section 3.7) in creating out improved method, we designed a distance metric to compare AOI renderings, and showed that the results of improved algorithm are closer to (good) human drawings than the results of the original outer-skeleton algorithm. This serves as a validation of the design choices made in the improved algorithm.

Section 4.2 presents the empirical evaluation conducted to compare the quality of computer and human drawings. Section 4.3 presents a quantitative comparison of the human-drawn and computer renderings. Section 4.4 presents and discusses the results of our evaluation. Section 4.5 concludes this chapter.

4.2 Empirical user-study

The purpose of this user-study is to deliver insight in how users perceive the quality of computer-drawn AOIs as compared to hand-drawn diagrams. Specifically, we want to assess which type of AOI rendering is the best, and why. Since we are not aware of specific studies to evaluate the quality of AOI renderings, firstly we shall consider the wider range of evaluating quality aspects of UML diagram renderings.

4.2.1 Related work

Related work concerns evaluating the quality of a visual depiction of system (UML) architectures. Purchase *et al.* have conducted numerous user experiments to assess the comprehensibility, aesthetics, and user preferences of UML (and similar) diagram renderings [74, 73, 75, 76]. Such results are valuable both as methodology and lessons learned, yet they cannot be applied directly to our problem, since AOIs are an extension of the standard UML notation. Several authors propose frameworks and methodologies to evaluate the comprehensibility and overall quality of UML models [47, 10, 3, 54, 66, 80]. Still, the question ”what are good quality criteria for visual modeling languages” is not exhaustively answered.

There is a large body of related work in the area of quality attributes of hand-drawn diagrams and diagram annotations beyond UML. Notably, Plimmer *et al.* have presented several systems for human annotation of computer-made diagrams [15, 69]. In particular, the RCA tool manages user-drawn annotations to fit around edited source code in an IDE, which is similar to our requirement that AOIs should fit the elements they enclose, regardless of their layout [72]. Beautification issues of hand-drawn diagrams and annotations are discussed by Plimmer and Grundy [68] and Yeung *et al.* [124]. Identified desirable issues such as annotation line smoothness, annotation constrainment to user-specified layouts, and the use of a natural stroke or flow-of-hand, are all directly relevant to our computer-drawn AOIs, as we discovered from the early phase of our AOI design process. Our particular challenge is, however, to generate such annotations entirely automatically, rather than starting from a user sketch.

One emerging conclusion from previous work is that plain, unannotated UML is often hard to comprehend and can perform better if extended by task-specific annotations. Our areas of interest are precisely such an annotation, useful to show cross-cutting concerns atop of a given system structure. Since this is a new notation, the characteristics that make for a good AOI drawing have not yet been studied in particular. Our aim is to construct a computer algorithm that renders AOIs similarly to good hand-drawn AOIs. The mentioned requirements mentioned in Table 3.2 attempt to capture the *a priori* quality criteria of a good AOI drawing - that is, what we, as visualization designers, think a good drawing should look like. Yet, to assess the *perceived* quality of an AOI drawing, we need a specific study. The next sections present such a study.

4.2.2 Experiment set-up

The empirical evaluation was designed and executed in the following way. Thirty users of higher computer science education levels (master, post-master, PhD, senior software designers, and computer science researchers) were selected. Some users were enrolled at the Eindhoven University of Technology in the Netherlands. Some others were part of an industrial European research project [40] where visualizing trust-related areas of interest on UML and component diagrams were a key task. This project is further discussed in Chapter 7. All users had good knowledge of UML and had worked before for at least a few months (up to a few years) with class diagrams in daily or weekly software design activities. The evaluation flow is depicted on Figure 4.1. It consists of three stages: drawing production, drawing comparison and results evaluation.

In the drawing production phase, the participants were given a complex class diagram with 110 classes marked by numbers, printed in black-and-white on an A4 paper (Figure 4.2) and seven AOIs, each given as a list of class numbers, printed on a separate paper. The list looked as follows:

- Area 1: 4, 5, 13, 12, 17
- Area 2: 51, 52, 55, 58, 59, 57, 68
- Area 3: 14, 21, 22, 32, 40, 41, 60, 58, 59, 80
- Area 4: 33, 34, 35, 36, 43, 61, 62, 66

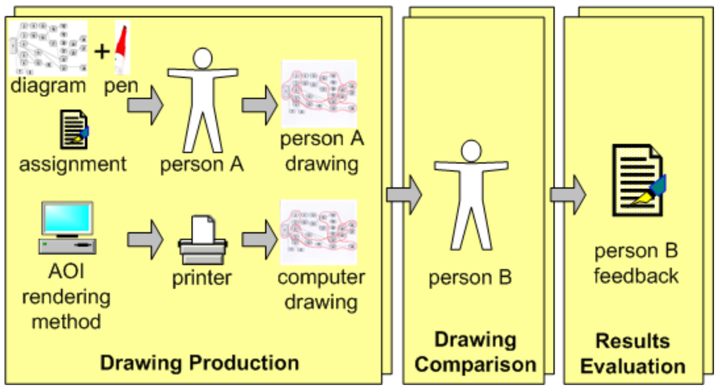


Figure 4.1: Evaluation process

- Area 5: 66, 82, 81, 95, 96, 103, 105, 104
- Area 6: 49, 50, 51, 67, 69, 111, 76, 78
- Area 7: 86, 92, 93, 99, 94, 80, 96, 95, 103

The participants were next asked to draw the areas as contours on this diagram, with a red marker pen we provided ourselves. The subjects were told that the goal of the drawing is to accurately and quickly convey, to another person, which class is in which area(s), and which area contains which classes. An example drawing, done on a different, much smaller, UML diagram containing 10 classes and one AOI, was also provided for basic illustration purposes. The complete experiment instructions were also provided on a separate A4 sheet. The subjects were given also a few paper sheets to practice on, before producing the final drawing. No verbal indications were given during the actual work, which lasted approximately 15 minutes. The subjects were not supervised. Also, they all worked independently, and had no knowledge of, or access to, the results of other participants.

Figure 3.24 a shows a scan of the drawing done by one of the participants.

Apart from the drawings made by the participants, and without their knowledge, we also produced a computer drawing on the same class diagram using the AOI rendering method described in Section 3.5. We adjusted the algorithm and rendering parameters (*e.g.* line thickness and color) to look as similar as possible to the human drawings, and then printed the computer drawing on a similar sheet of paper. The scan of the given computer-drawn AOI is shown in Figure 3.24 b. Essentially, the only salient difference between the computer and human drawings is the shape of the contour.

In the drawing comparison phase, we collected the results, and gave to each participant two drawings: a randomly picked drawing of another participant, as well as our unique computer-rendered drawing (Fig. 3.24 b). Without telling which is which and without giving any hint that one of the drawing was computer-made, we asked the participants to complete a questionnaire. The questions included:

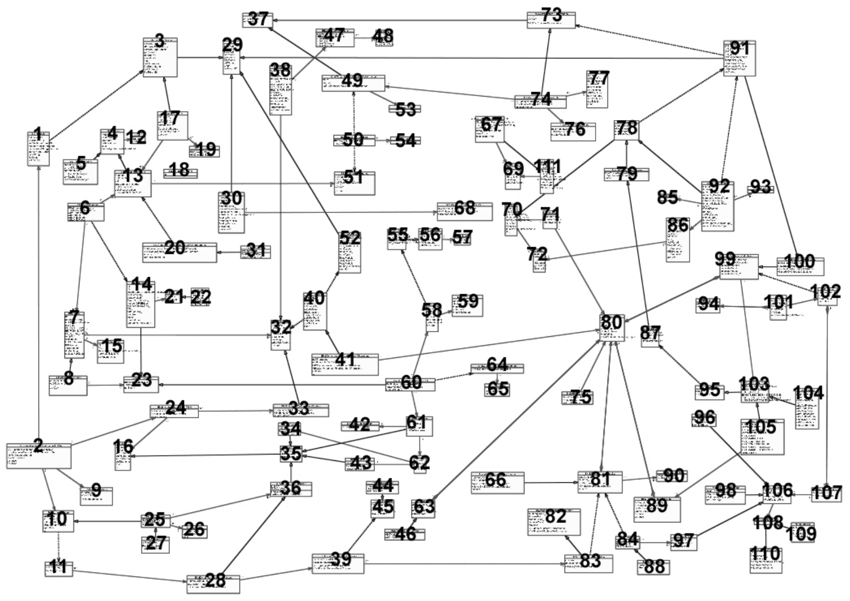


Figure 4.2: Class diagram used in the evaluation

1. rank the ease of understanding of the areas in each drawing on a scale of 1 (hardest) to 5 (easiest), accordingly to a Likert scale [53]
2. which is the most complex area to understand
3. rank the perceived similarity between the two drawings on a scale of 1 to 3
4. list, in plain text, what you liked least in the given drawings
5. list, in plain text, what you liked most in the given drawings

In the questionnaire, we mentioned that the main quality of an AOI drawing is given by its understandability, which is further related to its purpose. That is, the drawings should clearly show which area contains which classes, and which class is in which area(s). The questionnaire data was analyzed and aggregated. After collecting the questionnaires, we also had some short discussions (10-15 minutes) with the participants, in which we let them freely present their impressions and explain their results, and silently recorded their observations in writing. The results of the user study is discussed in the next section.

4.2.3 Evaluation results

The results of the questionnaire are summarized in the table in Figure 4.3. Several points become apparent now, as follows.

Most users found the machine-generated drawing (M) to be comparably understandable to the human-made one (H). Yet, the human drawings were almost always found to be better than the machine-generated ones, *i.e.* in 29 out of 31 cases (94%) (Figure 4.3, column A). The perceived similarity between the machine and hand-drawn AOIs (Figure 4.3, column D) showed a larger spread among users: 19 out of 31 (61%) marked a 2 ("not so different"), 10 users (32%) marked a 1 ("very different"), and the remaining two users (6%) marked a 3 ("very similar"). This can be explained by the relatively large variability of the different human drawings involved, and also in the fact that we refrained from providing similarity criteria to the users, to limit any potential biasing of the other assessments.

It is interesting to see that there is only a weak correlation between the perceived difference (column D) and the numerical difference between the perceived human and machine drawing qualities (columns B and C). Users that perceived their two drawings as being very different (value 1 in column D), *e.g.* rows 3, 5, 6, 8, 14, 17, 19, 22, 26, and 27 would score absolute human-machine quality differences of 1 (4 users, or 40%), 2 (2 users, or 20%), 3 (3 users, or 30%) and respectively 4 (1 user, or 10%). The two users that perceived their respective human and machine drawings ot be very similar (score 3 in column D) ranked their human and machine drawing qualities to be 4 and 2, and 4 and 4 respectively. Finally, the three users who indicated the highest human and machine drawing quality scores (5 and 4, respectively) all indicated a perceived difference of 2 between their drawings. Overall, the emerging impression is that similar drawings are not essentially implying the same drawing quality (from the perspective of the indicated AOI goals), nor would a similar quality in two different drawings imply that they perceptually look the same.

The hardest-to-grasp (most complex) areas were quite consistent, *i.e.* areas 2 and 3 (Figure 4.3, column E). This matches also our opinion, and gives further an indication that the drawings done by different users are of comparable quality concerning understandability. Among different drawbacks of the machine-drawn areas found during the results analysis, two were most frequently named. The first drawback concerns the *eraser* technique (Section 3.4.3). The eraser, used to mark elements overlapping an AOI contour but not logically part of that AOI, is not working well, as we indeed suspected beforehand. We call this the *wrong exclusion* problem. For example, class 56 is not part of Area 2, as it is wrongly suggested by the computer drawing. This is clearly visible in Figure 4.4, which shows a zoomed-in detail from the diagrams in Figure 3.24. Figure 4.4 a, drawn using the AOI rendering algorithm, does not show the AOI correctly. Figure 4.4 b, done by a human, is however correct. This problem was found by most subjects, as reflected in column G of the table.

The second drawback of the machine-generated areas concerns the contours' *tightness and smoothness*. These were perceived as being unpleasantly non-uniform (column F), and the *flow of hand*, *i.e.* similarity to the way humans draw, was lacking (column G). All users mentioned these aspects as hindering the drawings' understandability. As a third drawback, many subjects found the computer-drawn areas' *overlaps* confusing (column D). Contours which are near-tangent close to their intersection points were consistently named hard to understand in the light of the posed questions (Section 4.2.2). This was something we did not expect beforehand.

Clearly, there was room for improvement. The collected results point clearly in an overwhelming preference of the users for the human drawings, a fact which is also supported by the vast majority indicating higher or equal quality scores for the human drawings. It is natural to believe that a part of these differences are also reflected by the above-mentioned drawbacks of the machine drawings. In other words, removing some of these drawbacks has the potential of increasing the quality of the machine drawings. After analyzing the mentioned drawbacks, we designed several algorithmic improvements to the original AOI rendering method to address them. These improvements are in Sections 3.6.2 and 3.7.

Figure 4.5 b shows the result of our improved method on the same diagram detail as in Figure 4.4. We see that the improved method (4.5 b) is more similar to hand-drawing(4.5 a) than the original method (4.4 a).

4.3 Quantitative analysis

As the results of our user study showed (Section 4.2), the human-made drawings were perceived by users to be almost always better understandable than our computer-generated ones. We presented in Sections 3.6.2 and 3.7 several algorithm improvements by which we hoped to address the shortcomings of our computer rendering method.

However, how to measure how well we improved as compared to the original algorithm? Repeating the user study (Section 4.2) with the *same* audience could be biased, since the users by now knew our aims, diagram datasets, and already had some experience. Conducting the same study on a different group of subjects and/or different datasets could be done, but how to quantitatively compare subjective qualitative opinions of two different groups and/or datasets? Additionally, a user experiment does not give a precise, quantitative answer for how much closer or further our new algorithm improves the rendering.

If we were interested to test the suitability of the AOI renderings for a very specific comprehension task, *e.g.* the amount of time it takes to visually locate a given diagram element in a given area, we could indeed perform two user experiments to measure the time difference when using the improved, versus the original, rendering methods. However, there are several drawbacks to such an approach. First, as indicated by the user comments collected in our study, human-drawn AOIs have several quality attributes which help comprehension (*e.g.* the natural flow-of-hand). These are hard to quantify by means of *e.g.* timing a single (or a few) narrow tasks. We do not yet know which tasks would be the most representative here. Our main assumption is that drawing AOIs *as humans do it* is good for comprehension, in line with previous authors [68, 124, 72]. Hence, we would like to test that our improved algorithm produces drawings closer to human drawings than the original algorithm.

We designed a way which provides a quantitative answer to the above point. The quantitative analysis process is described next.

4.3.1 The quantitative analysis set-up

The analysis pipeline is shown in Figure 4.6 and described below.

Firstly, we extracted the area contours from all drawings, *i.e.* human and computed-generated with both the original and improved algorithm. For this, we used a simple filter-by-color thresholding technique, which was reliable as the contours and diagrams were drawn with two predefined distinct colors, *i.e.* red, respectively black, and we gave the users identical pens to drawn with. An example of the extracted contour is shown in Figure 4.7, which is indeed a clean, noise-free, contour representation.

Next, we would like to measure the difference between any two contours, *i.e.* human and/or computer-drawn. We do this as follows. Consider two contours C_i and C_j like the ones in Figure 4.7. For a contour C, we denote by D the distance transform, or *distance map*, of C. The distance map is defined as:

$$D(p) = \min_{q \in C} |p - q|, \quad \forall p \in \mathbb{R}^2 \tag{4.1}$$

for any point p in the 2D plane. Essentially, $D(p)$ gives the distance from any point p to the closest point q on the contour C. We know that the distance map of an object C is the solution of the so-called Eikonal equation:

$$|\nabla D| = 1 \tag{4.2}$$

with the boundary condition $D = 0$ on all points of C. We solve Equation 4.2 using the Fast Marching Method as described by Sethian in [84], on the same pixel grid as the one on which the scanned contour C is stored, and obtain the distance map D at a pixel-level spatial resolution. A careful implementation of the Fast Marching Method ensures the distance map D is computed on an image of 1024x768 pixels in under one second on a 1.8 GHz Windows PC [78]. Figure 4.8 shows the distance map D using a blue-to-red colormap (blue denotes low, red denotes high, distances) of the contour C (shown in white).

Now, given a contour C_i and its distance map D_i, computed as above, we define the distance d_{ij} of C_i to another contour C_j as:

$$d_{ij} = \frac{1}{2}\left(\frac{\sum_{p \in C_j} D_i(p)}{|C_j| D_{i_{max}}} + \frac{\sum_{p \in C_i} D_j(p)}{|C_i| D_{j_{max}}}\right) \tag{4.3}$$

In the above, D_j denotes the distance map of contour C_j, while $D_{i_{max}}$ and $D_{j_{max}}$ are the maximum values of D_i and D_j respectively over the considered images. $|C_i|$ and $|C_j|$ denote the contour lengths in pixels. The above definition of d_{ij} ensures d is a symmetric function $d_{ij} = d_{ji}$, and also is normalized between 0 and 1. Intuitively, Equation 4.3 states that the distance between the two contours C and C' is proportional with the area between the two contours, which is a perceptually good measure [18]. Alignment and image registration problems were, in our situation, not an issue, since all drawings were done on the precisely the same class diagrams, printed on identical A4 canvases, scanned at the same resolutions. Moreover, the distance metric given by Equation 4.3 is robust to small shifts and rotations. Let us stress that the proposed distance function is just one of the many ways in which we can compare contour drawings. Other more sophisticated measures, *e.g.* perceptual-based metrics [7], template-based matching [31], or contour matching using the earth mover's distance [35], can be used as well. We prefer to use our simpler metric since its numerical behavior is easier to interpret and its implementation is quite straightforward. Also, using more complex distance metrics typically involves having a clearer idea of which features (*e.g.* angles, protrusions, concavities, flat regions) are perceptually more important for the match, and how to quantify this importance. This is information that we do not have at the present moment.

4.3.2 Results of the quantitative analysis

We can build now a matrix d_{ij} containing all distances between any two contours of the 31 hand-drawn ones plus the two computer-drawn ones (with the original, respectively improved, methods). However, as the user evaluation results showed (Figure 4.3), not all hand drawings are found to be of the same quality. We are in particular interested to see how our computer-drawn contours compare to the good human drawings and, also, if the proposed improvements did, indeed, bring us closer to these drawings.

For this, we first split the 31 human drawings into three groups: good, average, poor, based the "human quality" scores of 5,4 and 3 respectively (see Figure 4.3). Figure 4.9 shows the distances of the six good drawings (H1..H6) to themselves and to the computer-generated drawings with the original (OLD) and improved (NEW) outer-skeleton methods. We see that all drawings in this table are quite similar to each other. We also see that the NEW drawing is consistently closer to the human drawings than the OLD drawing.

Figure 4.10 graphs the distances between all 31 human drawings and the two (initial and improved) computer drawings. The human drawings are grouped according to their perceived quality, as described above. Several observations can be made here. First, there is quite some variation in the distances within the same quality class. This is expected, since each quality value was assigned subjectively by just one person.

However, the distance values, averaged per quality class (Figure 4.11), show that the computer-generated drawings are *closer* to good than to bad drawings, both for the original and improved methods. Also, we see that the improved method brings the computer-generated drawings closer to the human drawings in all quality classes as compared to the original method. The improvement is of roughly 20 percent for the "good" and "average" classes and 12 percent for the "bad" class. This is also visible from the graphs in Figure 4.10. More importantly, the improved method brings the computer drawings closer to the good human drawings than to the bad ones.

Finally, we notice an outlier: the human-drawing 30 in Figure 4.10. Looking at it (Figure 4.12), we can indeed see it has a very different style from a typical good human drawing (*e.g.* Figure 3.24. Also, the user who made drawing 30 forgot to draw one area (Area 6 - see Section 4.2), which explains the high spike in the distance metric. This shows that our distance metric is quite discriminative in the presence of erroneous drawings.

4.4 Discussion

4.4.1 Human-machine drawing comparison

The distance metric proposed to compare contours is well-known in shape analysis and computer vision applications (see *e.g.* [18]). Its main advantage is good robustness to small-scale geometric noise, and rotation and scale invariance. However, it does not take into account specific quality attributes for the tasks related to areas of interest. For example, we can argue that a small geometric difference between two contours is perceptually more important if located at some point where several contours overlap or intersect, or in an area covered by a complex diagram, than at the periphery of the drawing. Integrating perceptually driven distance metrics [7, 31] in our evaluation should lead to further insights.

Finally, we are aware that we have not conducted a formal user experiment, *i.e.* a quantitative measurement of the (in)validation of a hypothesis. Our evaluation's main goal was to harvest information about the differences perceived between computer- and human-drawn AOIs, and to adapt our computer drawings accordingly. Assuming that our hypothesis holds that human-drawn AOIs are easy to understand, we argue that our improved rendering algorithm produces better drawings, since these are measurably closer to human drawings than the original computer drawings.

The main advantages of the machine drawings, as compared to the human drawings are as follows

- rapid and automatic handling of complex areas on large diagrams. A human will typically take several minutes to draw a set of areas, while the algorithm presented here needs a few seconds;
- correct drawing. As shown by the experiment, humans can draw incorrect areas, especially for complex configurations (see *e.g.* Figure 4.4);
- easy parameterization of the drawing process (contour smoothness, color, thickness, filled or not)

The main elements which still need to be incorporated in the automatic algorithm, as outlined by the user study, are

- contour *crossings:* Although the improved algorithm presented here significantly reduces sharp angles and produces overall smooth contours, it can still produce contour crossings having small angles. These crossings can be detected, and an additional force can be added to the contour points in the crossing's vicinity to maximize these angles in the shrinking process;
- contour *separation:* Since contours are drawn independently, they can have (near) overlapping fragments, which are hard to separate visually. This could be addressed by a separate relaxation pass: After all contours are constructed independently, repulsion forces are added to the contour points, and several deformation steps are performed. Alternatively, the contours can be drawn iteratively, and the repulsion forces can be added to each newly drawn contour as it is shrunk.

4.4.2 Limitations learned from the user study

Although the improved AOI drawing method yields better appreciated drawings, which are measurably closer to human drawings than the original method, it still has some limitations. First, our users have found near-tangently intersecting contours to be hard to understand (Section 4.2.3). We have not addressed this problem yet. For this, we should consider a global contour rendering rather than the per-contour rendering we use now.

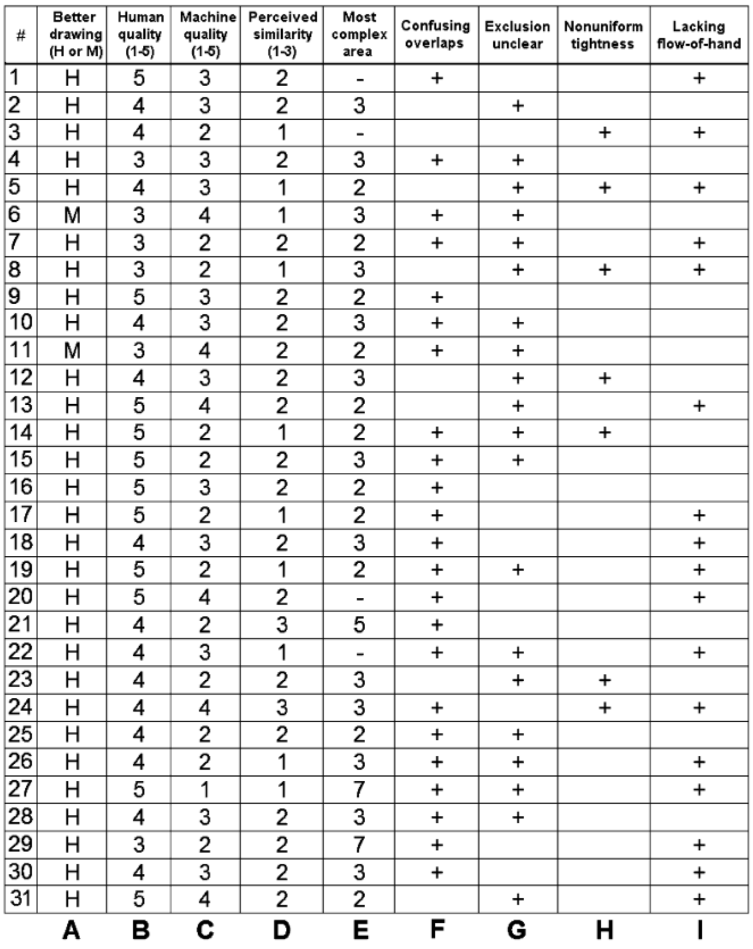
Besides a possible optimization of the contour crossing angles, a global contour construction could also optimize the actual positions of contour points, in order to pull apart AOI fragments which are too close. Although this direction should be explored in future work, it introduces some important problems. In typical usage scenarios, one neeeds to switch areas on and off interactively during analysis. If the shape and position of an AOI are influenced by other AOIs, then toggling on or off an area A_i may abruptly change the look of another area A_j in the same drawing, which is highly disruptive. Precomputing all AOI contours beforehand is not an option, since a typical diagram can easily have tens of areas showing different concerns (performance, resource usage, reliability, coding and documentation aspects, and so on), and we do not know in advance which areas one may want to visualize at a time. Finally, a global optimization may incur performance problems on large diagrams with many areas.

4.5 Conclusion

The main result of our evaluation can actually be seen as an *a posteriori* justification of the early decisions taken when designing the AOI rendering method based on outer skeletons. Based on the actual evaluation, we identified which of the drawing aspects are found important by actual users when completing typical understanding tasks, and discovered some limitations of our original AOI rendering method. Based on this insight, we designed several algorithmic improvements to the rendering method and showed quantitatively that our improvements bring the computer drawings closer to human drawings identified as good. Overall, this gives us good ground to claim that the improved AOI method is able to produce drawings which are, on the average, comparable in quality and understandability with good human drawings.

However, this study also identified several limitations of our AOI rendering method as compared to human drawings. The most important is our choice to draw areas separately, while humans consider already-existing information when adding new elements to a drawing. An instance hereof is our inability to optimize angles at contour crossings, while humans seem to do this when drawing overlapping areas.

With this chapter, we conclude our study of drawing areas of interest on software diagrams. The following chapters are dedicated to adding different types of metric information to diagrams annotated with areas of interest.



#	Better drawing (H or M)	Human quality (1-5)	Machine quality (1-5)	Perceived similarity (1-3)	Most complex area	Confusing overlaps	Exclusion unclear	Nonuniform tightness	Lacking flow-of-hand
1	H	5	3	2	-	+			+
2	H	4	3	2	3		+		
3	H	4	2	1	-			+	+
4	H	3	3	2	3	+	+		
5	H	4	3	1	2		+	+	+
6	M	3	4	1	3	+	+		
7	H	3	2	2	2	+	+		+
8	H	3	2	1	3		+	+	+
9	H	5	3	2	2	+			
10	H	4	3	2	3	+	+		
11	M	3	4	2	2	+	+		
12	H	4	3	2	3		+	+	
13	H	5	4	2	2		+		+
14	H	5	2	1	2	+	+	+	
15	H	5	2	2	3	+	+		
16	H	5	3	2	2	+			
17	H	5	2	1	2	+			+
18	H	4	3	2	3	+			+
19	H	5	2	1	2	+	+		+
20	H	5	4	2	-	+			+
21	H	4	2	3	5	+			
22	H	4	3	1	-	+	+		+
23	H	4	2	2	3		+	+	
24	H	4	4	3	3	+		+	+
25	H	4	2	2	2	+	+		
26	H	4	2	1	3	+	+		+
27	H	5	1	1	7	+	+		+
28	H	4	3	2	3	+	+		
29	H	3	2	2	7	+			+
30	H	4	3	2	3	+			+
31	H	5	4	2	2		+		+
	A	B	C	D	E	F	G	H	I

Figure 4.3: Results of drawing comparison. Columns A-E indicate: which drawing was overall found to be better (human or machine); perceived quality of the human drawing; perceived quality of the machine drawing; perceived similarity of the two drawings; visually most complex area. Columns F-I indicate often-perceived drawbacks of the machine drawings.

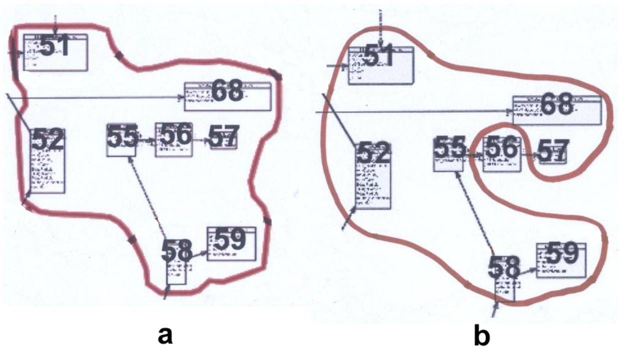


Figure 4.4: Element 56 is not part of the area, as correctly shown in the human drawing (b). The eraser-based technique incorrectly shows 56 as inside the area

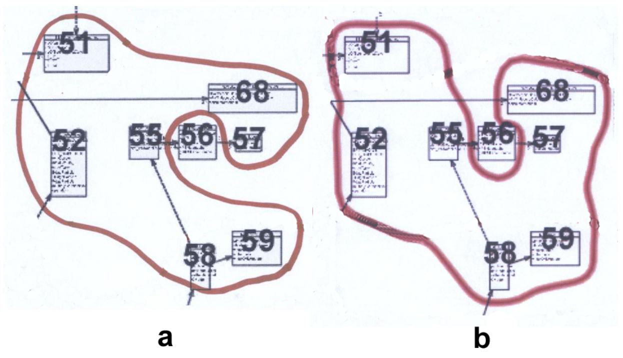


Figure 4.5: Comparison of human drawing (a) with the improved rendering method (b). Details from Figure 3.24)

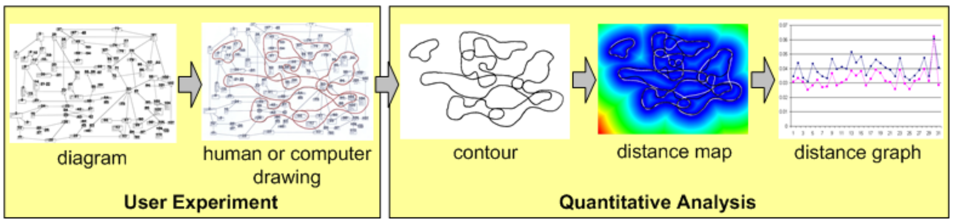


Figure 4.6: Quantitative analysis procedure

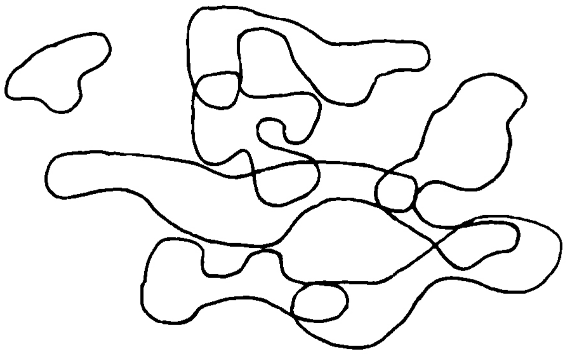


Figure 4.7: Extracted contour

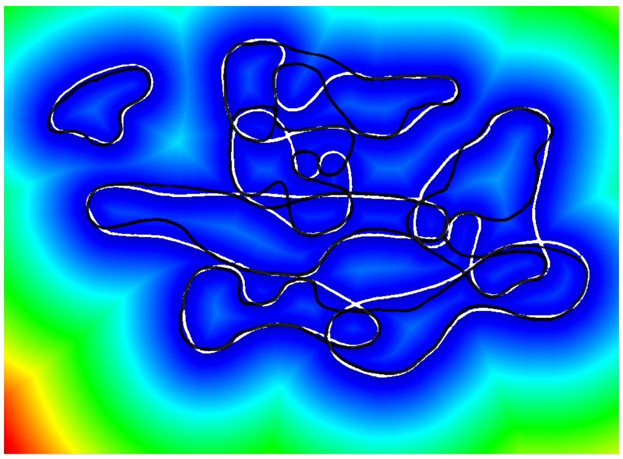
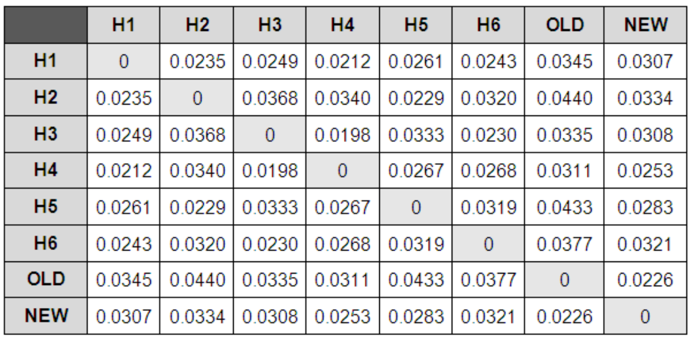


Figure 4.8: Distance map D (blue-to-red colormap) of contour C(white), used to compare C with a second contour C' (black).



	H1	H2	H3	H4	H5	H6	OLD	NEW
H1	0	0.0235	0.0249	0.0212	0.0261	0.0243	0.0345	0.0307
H2	0.0235	0	0.0368	0.0340	0.0229	0.0320	0.0440	0.0334
H3	0.0249	0.0368	0	0.0198	0.0333	0.0230	0.0335	0.0308
H4	0.0212	0.0340	0.0198	0	0.0267	0.0268	0.0311	0.0253
H5	0.0261	0.0229	0.0333	0.0267	0	0.0319	0.0433	0.0283
H6	0.0243	0.0320	0.0230	0.0268	0.0319	0	0.0377	0.0321
OLD	0.0345	0.0440	0.0335	0.0311	0.0433	0.0377	0	0.0226
NEW	0.0307	0.0334	0.0308	0.0253	0.0283	0.0321	0.0226	0

Figure 4.9: Distance table of the best human drawings and AOI rendering

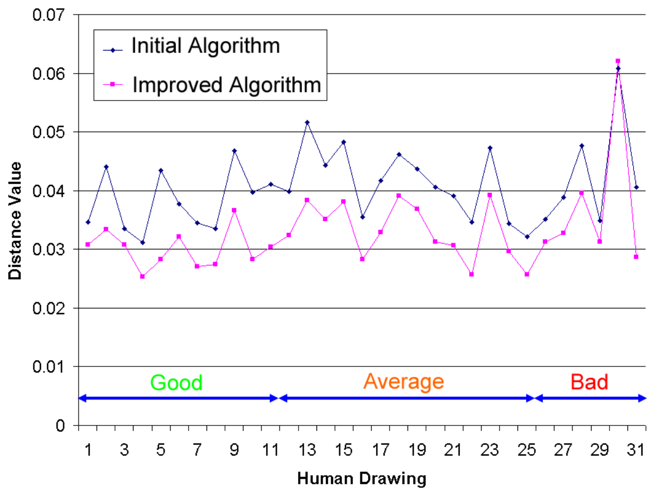


Figure 4.10: Comparison of AOI rendering algorithms. The improved rendering method yields results closer to the human-drawn AOIs than the original rendering method

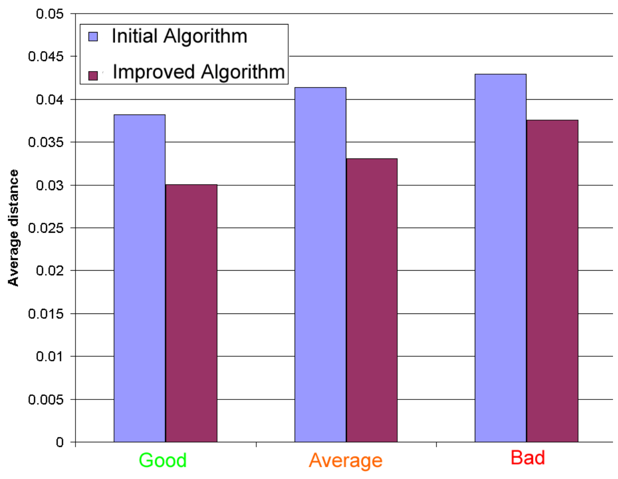


Figure 4.11: Comparison of average distances between three groups of human drawings (good, average, bad) and two computer-generated drawings (initial and improved).

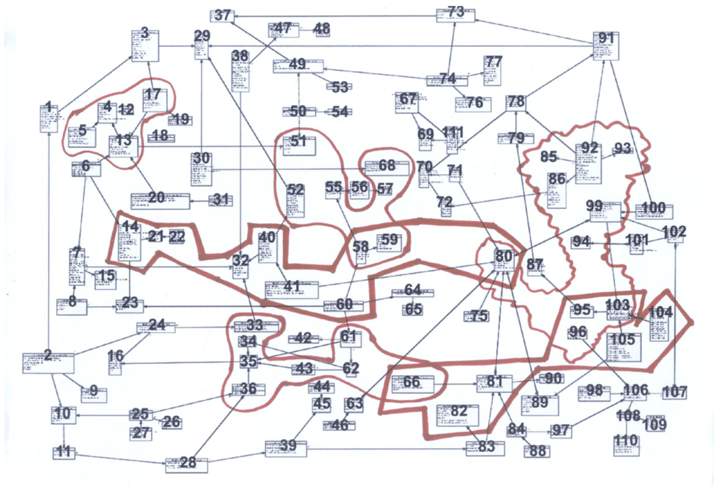


Figure 4.12: Atypical human drawing (a scan of the paper image)

Chapter 5

Method-level metrics

In this chapter, we address the question of how to visually correlate an object-oriented software architecture with metric values defined at *method* level. To support this task and its specific scalability and understandability requirements, we designed two new visual metaphors. First, we propose the metric lens, a new visualization of several method-level code metrics atop of traditional UML class diagrams, which allows users to quickly perform metric-metric and metric-structure correlations on large diagrams, and also works in a level-of-detail manner. Second, we present the metric legend, a new way to interactively specify a spectrum of analyses involving different metrics, value ranges, and visual mappings. The metric legend is interactively correlated with the metric lens to enable users to quickly specify a wide palette of analyses, and also acts like a legend for the data displayed in these analyses.

5.1 Introduction

There are several possible sources for the architectural diagram data which has been the subject of the previous chapters. At a high level, we can classify these sources in two main categories, according to the software development life-cycle introduced in Chapter 1:

- *design-phase diagrams:* these diagrams are manually specified during the design phase of a system;
- *maintenance-phase diagrams:* these diagrams are typically extracted from source code in the maintenance phase.

Diagrams in the above categories share the same data: entities, relationships, areas of interest, and various metrics. Their main role is also quite similar: to convey insight into the high-level structure and function of a given system. In this context, one way to understand source code is to represent it on a higher level of abstraction, *e.g.* on a design or architectural level. If someone would like to analyse an object-oriented system, UML diagrams would be the conventional choice to represent the system on a design level. However, there are also several differences. Design-phase diagrams have, typically, a weak relation with lower-level artifacts such as source code, since these artifacts do not yet exist in the design phase. Diagrams created during maintenance, however, must take the existing source code into account, as source code is often the most important asset produced during software development, also called the ”hard currency” of software development [94].

During maintenance, diagrams are put into accord with source code typically by extracting them directly from source code in a process called reverse engineering or reverse architecting [46, 110]. This creates several challenges, which we can relate to our original requirements of *scalability* and *understandability* (Chapter 1). First, we need effective methods to present, or visualize, extracted diagrams, much in the same way we did it with design-phase diagrams. Second, such methods need to be scalable. Diagrams extracted from source code are, often, much larger and more verbose, *i.e.* have more entities, relationships, and class members, than hand-drawn diagrams in the design phase. The reason is that it is very hard to automatically separate low-level (implementation) details from higher-level (design and architecture) facts when extracting diagrams from source code.

Since diagrams produced from source code are quite large, we need ways to simplify these in order to achieve our understandability goals. Methods to do this can be classified in two categories, depending on the moment when they occur:

- *analysis methods:* the data extracted from source code is simplified, of abstracted, before its (visual) presentation;

- *visualization methods:* the extracted data is simplified, or presented in a scalable way, during the visual presentation.

In the first class of methods, several approaches are possible, such as automatic techniques for extracting and filtering design patterns from code [59, 88], and metrics-based techniques [51]. Metrics-based techniques are particularly interesting in the context of our work. These techniques abstract the low-level structure of the source code by reducing it to a number of metrics that capture properties of interest, such as complexity, testability, maintainability, and evolvability [51, 104]. In this thesis, our interest goes to the second class of *visualization* methods. As such, we shall not concern ourselves with ways to improve the source-code mining process, or with specific requirements for these methods such as scalability, accuracy, and completeness. Our assumption is that we have a set of software diagrams extracted from source code, together with several code-level metrics computed at class or class-member level, and we wish to present these visually in a scalable and understandable way[1]. This is the focus of this chapter. In the next chapter, we address the related problem of visualizing several types of metrics computed on groups of diagram elements, or areas of interest.

Combining code metrics and diagram information in a single representation is an effective way to help several types of system assessments, such as spotting (cor)relations among code attributes, relations, and diagram element types; determining where, on a system’s architecture, do code attributes reach outlier values; and identifying specific

[1]In this chapter, we shall use the terms *method* and *member* interchangeably, since we treat both artifacts identically from the perspective of visualization

code patterns and their correlation with code metrics. Ultimately, these activities help users to assess the quality, modularity, and maintainability of a given software system.

To be most effective, a solution for combining code metrics and structural (diagram) information should comply with several requirements, as follows:

1. show the system structure and code metrics in a *single* representation, so that metric-structure correlations are easy to do;

2. show *several* metrics in the same time, which may have different value ranges, so that metric-metric correlations are easy to do;

3. the envisaged solution should be *scalable* for large diagrams of tens or hundreds of classes having hundreds of methods, and several metrics;

4. let users specify and control all above operations in an easy, *interactive*, way.

These requirements map directly on our original requirements of UML-based presentation, scalability, and understandability (Chapter 1). Summarizing the above, we aim at a technique that should enable correlation and comparison between many metrics, having different ranges and meanings, on large diagrams, all in an easy, interactive way.

In this chapter, we propose such a solution, consisting of two new combined techniques: the *metric lens* and the *metric legend*. We start by overviewing the improvements which can be done in the areas of software architecture and software metrics visualization (Sec. 5.2). Section 5.3 outlines activities that come before the code metric visualization, *i.e.* structure and metrics data extraction from source code. Section 5.4 presents our new solution for structure-and-member-metric visualization: the *metric lens*, which adapts and extends the well-known table lens technique for software datasets (Sec. 5.4.1); and the *metric legend*, which lets users both specify analyses interactively and interpret the metric lens visualizations (Sec. 5.4.2) and outlines the implementation of our techniques (Sec. 5.4.3). Section 5.5 presents case studies of our techniques in understanding a real-world software system. Section 5.6 discusses our results. Finally, Section 5.7 concludes the chapter.

5.2 Related work

In this chapter, we focus on a specific class of software: object-oriented (OO) code bases written in an OO language, such as C++, Java, or C#. OO software is highly structured, a favorable attribute in development, maintenance, and understanding in general. However, OO software can also be highly complex, posing an increased burden on understanding, due to the many types of relationships that can be created at code level, *e.g.* the many flavors of data and type dependencies, and complex inheritance hierarchies. Although several methods and tools exist for understanding this type of software, there are still limitations in the provision of a high integration of data mining and data presentation (visualization) capabilities, which limit their applicability.

The related work in the areas software architecture and software metrics visualization is reviewed in Section 2.2. As already outlined there, the above methods can be improved in several directions. First, mapping software metrics to visual attributes of diagram nodes, such as size, color, texture, and shading, works well for metrics defined at the level of an entire class (or component, for component-based diagrams) but does not scale to the finer-grained level of *method* metrics. Adding metric icons to UML-based visualizations [107] has the advantage of using an accepted structural visualization (UML), but cannot show method-level metrics.

Secondly, in all the systems we are aware of, the process of choosing and correlating a subset of the several available metrics is quite difficult for non-experienced users. In a typical reverse-engineering process, many metrics can be computed on hundreds of methods of the classes of the same system, *e.g.* code complexity, coupling, cohesion, fan-in, fan-out, and comment density. The question is: how to assist the user in the process of specifying how to map these metrics to visual attributes in a given analysis scenario, such that the visual correlation of the selected metrics is easy to do and supports the questions at hand.

In the remainder of this chapter, we shall address the two goals mentioned above, *i.e.* designing a scalable and understandable visualization of method-level metrics on UML diagrams; and supporting the user task of visually choosing and correlating different such metrics.

5.3 A software analytics pipeline

As outlined in the previous sections, our aim is to have an *end-to-end* solution that seamlessly combines data mining and visualization for object-oriented software systems, all starting from the source code. Hence, we must consider several issues:

- how to extract metrics and structure from source code;
- how to visualize these in a scalable way;
- how to enable users specify the question they are interested in an easy, interactive way.

Our proposed solution consists of a pipeline of operations, as outlined in Fig. 5.1.

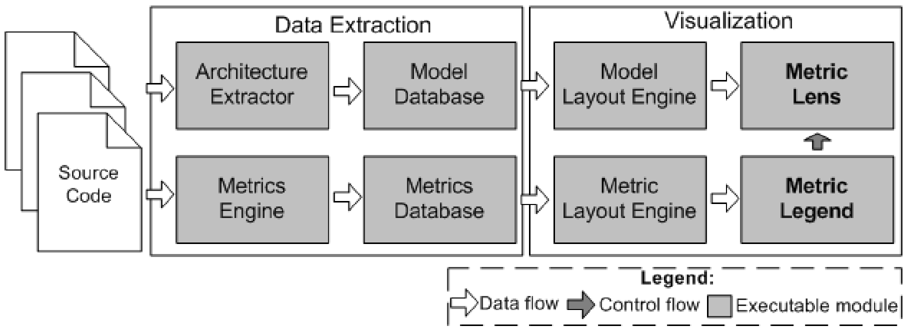


Figure 5.1: Architecture of the metrics-and-structure visualization pipeline

The pipeline is divided in two major components: data extraction and data visualization. Hence, our proposal combines both *data mining* and *data visualization* in a single solution. This is an exact instance of the so-called *visual analytics* process: combining data processing and mining with interactive visual interfaces for the goal of analytical reasoning about a given system [109, 121]. In this context, we call our approach *software analytics*: the combination of structure-and-metrics data mining and visualization techniques for understanding software maintainability, which explains the title of this section.

5.3.1 Structure and metrics data extraction

Our visualization target, in the terminology of [58], is a system model, consisting of a set of UML class diagrams D_i. For each class in such a diagram, we store its methods and data members, as well as a number of different real-valued software metrics $\mu_1, \ldots, \mu_n$, each having a given value-range $\mu_i \in [m_i, M_i] \subset \mathbb{R}$. This model fully complies with our general data model discussed earlier in Sec. 3.2 and represented in Fig. 3.1.

Since all our input is the software source code of the system under study, we must extract our UML model from this code. For the applications presented in this section, we shall focus on code bases written in the C++ programming language [94]. However, we note that the visualization techniques which form the subject of this chapter are directly applicable to diagrams and metrics extracted from any object-oriented languages, once one avails of the needed data extraction tools.

For the task of extracting structure from source code, we can use various tools, as follows.

First, we can use the CCCC extractor [55]. CCCC is a so-called *ligthweight* extractor. Lightweight extractors do not attempt to parse all constructs in the input source code, but focus on the subset of these constructs which is relevant to the type of information they aim to collect. As such, they are relatively simple and fast, as they do not have to implement the full grammar of the language under analysis, nor do they have to perform involved operations such as type checking and disambiguation. Code constructs which are not understood in the input, either due to the extractor's parsing limitations, or due to inherent incorrect or incomplete code in the input, *e.g.* code with missing headers, are skipped. The result emitted a lightweight parser is a partial partial abstract syntax tree (AST) of the input code.

A second solution for the structure extraction from source code is to use a so-called *heavyweight* extractor. In contrast to lightweight extractors, heavyweight ones attempt to process all code constructs in the input and produce a complete AST. For heavily context-dependent languages such as C++, this often involves also performing type checking (semantic analysis) of the input, as several C++ constructs cannot be correctly parsed without this information [67]. Several heavyweight extractors can be used for our task, *e.g.* Columbus [29], SolidFX [104], and extractors constructed with the ANTLR parser generator [67] using the C++ grammar described in [120]. The advantage of such tools is that they produce the entire AST of the input code, thereby allowing the extraction of many more types of code relationships than lightweight parsers, *e.g.* declaration-definition relations, variable or function use-definition relations, or template information. Depending on the aspects we want to capture in our UML diagrams, such relations may be necessary.

A second advantage of heavyweight parsers is that they can correctly handle the extraction of relations in complex code, *e.g.* declaration-definition relations that depend on the complex lookup rules of the C++ language, which in turn require semantic analysis. The disadvantage of heavyweight extractors is that they are usually significantly slower than lightweight ones and require a more complex set-up phase. For example, an ANTLR-based parser for C++ is up to 5 times faster than SOLIDFX, but produces less detailed information, *e.g.* will skip the code in a class or method declaration after a parse error up to the end of the enclosing scope. SOLIDFX is much more robust, but slower. A detailed discussion of the relative advantages and disadvantages of heavyweight and lightweight extractors for C++ can be found in [44].

After the basic classes and class-relationships are extracted from the source code, we must specify how to group classes and relationships into *diagrams*. As outlined in Chapter 1, each diagram in a UML model typically captures a different concern, aspect, or subsystem of a given design. However, such information is, in most cases, not explicitly present in the source code, so we have to add it back when extracting the diagrams. The solution we used here was for the user to manually specify a division of the analyzed code base into subsystems by assigning source files to each subsystem. Next, one UML diagram for each subsystem is automatically created, containing the classes in that subsystem's files. Alternative methods could be used, *e.g.* automatic class-to-diagram and relation-to-diagram assignment, in those cases when (meta-)information is available to this end.

The second step of the data extraction from source code concerns the computation of software metrics. We can compute code quality metrics both on the level of classes and class methods. Class metrics include the typical number-of-methods, number-of-bases, inheritance-depth, and number-of-overriden-methods [51]. Method metrics include the lines-of-code (LOC), lines-of-comment-code (COM), and McCabe's cyclomatic complexity (MVG) metrics, among others. In the following, we shall focus on the method-level metrics, since it is for these that we provide our novel visualizations (Sections 5.4.1 and 5.4.2).

For metrics calculation, we used the Understand software analysis tool [82], which can compute all the method-level metrics mentioned above efficiently and reliably on large C++ code bases. The extracted metrics are saved in a XML-based database, where each row corresponds to a class or method, and each column to a particular metric. Depending on the completeness of the extraction process, the values of several software metrics can miss for specific methods. We will have to accommodate these missing values in our metric visualization (Section 5.4.1).

Separating the structure and metric extraction processes allows us to easily upgrage our metrics collection by adding new parsers, rather than attempt to achieve everything within a single tool. This massively reduces development effort. Overall, we adopt here a pragmatic approach, similar to [8]: we use a combination of existing tools, whenever available, to extract and combine the complementary information needed for our final analysis.

After extracting the UML model and method-level metrics, this information is next fed to the visualization step of our software analytics pipeline. This step, and the two novel techniques it introduces, are described next.

5.4 Visualization Design

Our combined structure-and-metrics visualization design revolves around two views: the *metric lens* and the *metric legend*. The views are tightly coupled, as shown in Figure 5.2. The metric lens combines a classical UML viewer with a visualization of method-level metrics using an enhanced version of the well-known table-lens technique [77]. The metric legend has the double role of allowing users to specify which metrics, value ranges, and visualization metaphors to use in the creation of the metric lens view, and of acting as a legend for the displayed metrics in this view.

As mentioned in Sec. 5.1, the goals of our two-view solution are:

- specify and show several method-level metrics simultaneously
- correlate and compare several metrics
- easily spot outlier metric values
- emphasize metric values in a specific range

The following sections describe how these goals are met by our two-view solution, *i.e.* the metric lens (Sec. 5.4.1) and the metric legend (Sec. 5.4.2).

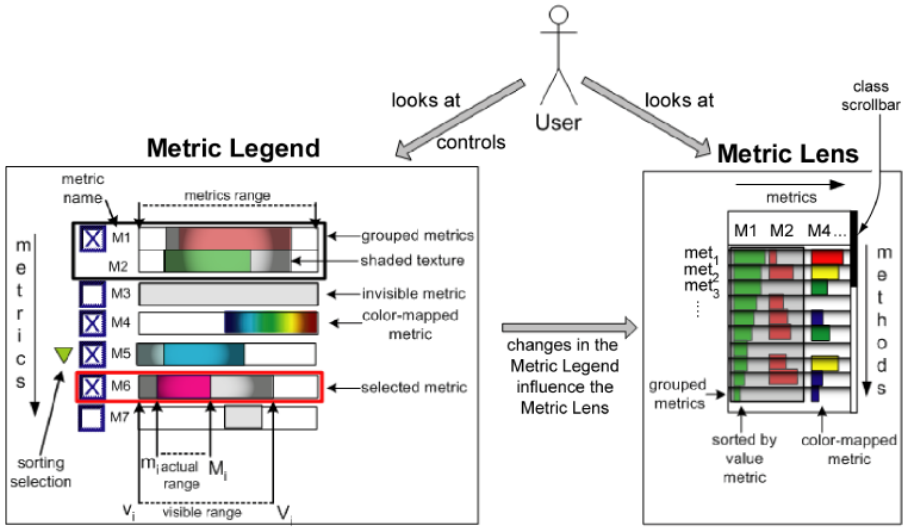


Figure 5.2: Stucture-and-metrics visualization design: metric legend (left) and metric lend (right)

5.4.1 Metric lens visualization

The basis of the metric lens technique is a traditional UML class diagram, which displays all data members within each class frame (see Fig. 5.3). Atop of this image, we display metrics following a *table model*, where the rows are methods met_i and columns are metrics M_j. Each table cell shows one metric value using different icons. Missing metric values (Sec. 5.3.1) have no icon. All metric tables of all displayed classes can be sorted on various criteria, such as the method names or metric values, enabling different types of analyses, as discussed next in Section 5.5. The metric icon table can be placed within the class frames (Fig. 5.3 a,b), which yields a compact layout but does not let users read the method names, or on the right side of the class frames (Fig. 5.3 c), which does not occlude the displayed method names but yields a less compact layout.

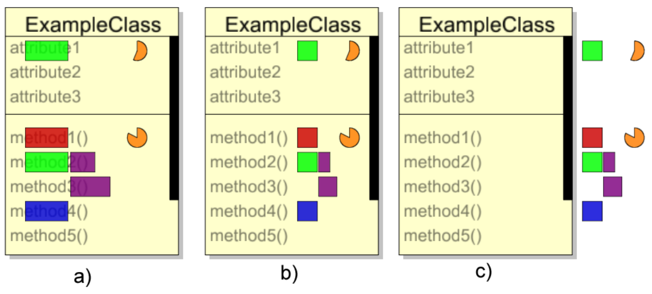


Figure 5.3: Metric layout options (actual snapshots)

Each metric value in a table cell is shown using a metric *icon*. We first used here the same design as in [107], *i.e.* a number of general-purpose icons such as 2D and 3D bars, pie charts, and cylinders, scaled by the metric values. However, this design does not scale well, since a UML diagram can easily have a few hundred methods, several tens per class, and each such method can have several metrics. In such a case, the actual screen space that we can dedicate to display one metric icon can be very small, *e.g.* under 10 pixels.

To address this problem, we use a modified version of the table-lens technique [77]. We start by rendering each table cell as a horizontal bar, scaled and/or colored by its metric value. The actual value-to-color or value-to-size mapping is controlled by the metric lens widget, described next in Section 5.4.2. Along with this, we provide two independent zoom mechanisms:

- *diagram-level:* this zooms the entire diagram, *i.e.* both the UML layout and the metrics and method names of each class;
- *class-level:* this zooms only the contents of each class (metrics and method names) but keeps the UML layout fixed.

The two zoom modes serve different purposes. Diagram zooming allows users to focus on a specific subsystem, *i.e.* displays a subset of the entire diagram at high resolution, so one can read the actual method names shown within the displayed classes. Class-level zooming, in combination with class contents sorting, allows users to smoothly navigate between seeing the the entire contents of each class, as a set of colored bar graphs (when zoomed out), and seeing the individual signatures and names of methods and members (when zoomed in). When the class-level zoom level is low and the class frame size cannot accomodate all methods and metric icons, we display a scrollbar at the right of each class and allow users to scroll through its contents (see Fig. 5.2 right). We also implemented two enhancements as compared to the original table-lens method [77]: First, we modulate the methods' text opacity by the class zoom level, so that class-zooming effectively smoothly interpolates between a traditional text-based diagram view and a set of bar graphs. Second, we render each metric bar using a vertical gradient-shaded (dark to bright) texture (see Figure 5.2 right). This creates a subtle contrast that visually separates the extents of each method when the class contents are zoomed out and the text is not visible, thereby allowing one to better distinguish the individual metric icons.

5.4.2 Metric legend

When visualizing several different metrics over large UML diagrams, inferring the exact metric values from the sizes and/or colors of the metric bars can be very hard. Moreover, additional questions remain: How to visually compare metrics which are defined over totally different value ranges, such as *e.g.* coupling and lines-of-code; and how to specify which metrics to compare in a given scenario from the potentially large available set of metrics produced in the extraction phase? (Section 5.3.1)

We address these goals using a new visualization: the *metric legend* widget. This widget has two roles. First, its lets users interactively control which metrics from the data model are mapped to the icons of the metric lens, and how (Section 5.4.1). Second, it acts like a legend for interpreting the icons in the metric lens visualization.

Basic design

The metric legend has a compact tabular structure (see Figure 5.2 left for a schematic overview and Figure 5.4 from an actual screen snapshot). Each metric μ_i of the data model generated by the metric extraction phase occupies a row in the widget. For each such metric, a checkbox in the legend shows whether it is *visible*, *i.e.* shown with colored bars in the metric lens, or not; the metric's *name*, *e.g.* $M1 \ldots M7$ in Figures 5.2 left and 5.4; and the metric's actual and visible *ranges*. To explain the latter two, consider a metric having values in the range $[m_i, M_i] \subset \mathbb{R}$. The right part of the metric legend in Figure 5.4 displays the ranges $[m_i, M_i]$ of all metrics μ_i as colored bars, scaled and translated so that we can compare them visually. For example, we see that the maximal values of $M3$ and $M4$ are the largest among all available metrics, and that the range of $M5$ is contained in the range of $M6$.

The bar colors indicate how each metric is shown in the table lens, as follows. Gray denotes metrics that are not shown in the table lens, *e.g.* $M3$ and $M7$. A flat, uniform, color indicates that the bar-icons of that metric in the metric lens are simply drawn in that color. This mode is useful when we want to encode metric values in the bar sizes, and metric identities in the bar colors, *e.g.* $M1, M2, M5$ and $M6$ in Figure 5.4. A blue-to-red (rainbow) colormap indicates that the respective metric is hue-mapped in the metric lens using colors from blue (indicating the minimum m_i) to red (indicating the maximum M_i), *e.g.* $M4$. Other colormap choices are, of course, possible. Clicking on the legend widget allows changing the color mapping from flat to rainbow.

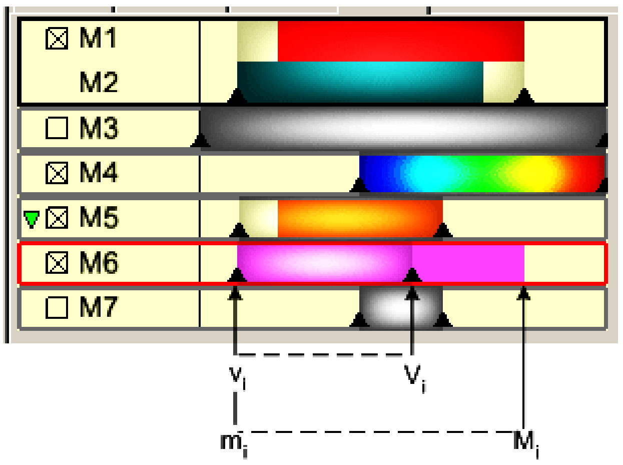


Figure 5.4: Metric legend (actual tool snapshot)

Selecting visible metric ranges

In most cases, software metric values are not uniformly spread over their actual ranges $[m_i, M_i]$. Values may be concentrated in, say, the lower range half, with only a few spurious values in the upper half. In such a case, we do not actually want to visualize the entire actual range $[m_i, M_i]$ of that metric, but only the lower half, where the most values are. The metric legend supports this by specifying a so-called *visible range* for each metric, *i.e.* an interval $[v_i, V_i] \subset [m_i, M_i]$. Users can specify the visible range by dragging two sliders (show as small black triangles in Figure 5.4) over the range bar of the desired metric in the metric legend. Values outside the visible range $[v_i, V_i]$ are clamped to v_i (if lower) and respectively V_i, if higher. This effectively lets users focus the metric visualization over a desired subrange of values, with direct applications, as shown later in Section 5.5.

As mentioned above, one of our goals was to create compact visualizations. As such, the metric legend can contain many different metrics, each with a different visible range. We noticed, during actual use of the widget, that visually comparing the different ranges in this widget by comparing the positions of the small triangular sliders is quite hard. To enhance the perception and support the visible comparison, we show visible ranges by blending a half-translucent shaded texture, dark at the margins and bright in the center, atop of the colored range bar, between the two sliders. This emphasizes that part of the actual range which is visualized, without obscuring the color map drawn in the bar. For example, in Fig. 5.4 we see that the visible range $[v_i, V_i]$ for metric $M6$ is approximately the lower half of its actual range $[m_i, M_i]$, by comparing the sizes and positions of the respective shaded textures.

We construct these shaded textures as follows (see also Figure 5.5). First, we build a parabolic luminance texture dark at the borders and bright in the center, similar to the so-called structure cushions used by [56], and multiply the actual color-mapped metric with it. This effectively darkens the metric’s color at the borders and keeps it unchanged in the center. Next, we blend the result with a white rectangle, textured with a Gaussian-shaped transparency texture opaque in the center and transparent at the borders. The final result shows a metric bar with specular-like highlight in the center, and dark at tbe borders. Using textures to mark subranges atop of an existing visualization is more effective, and visually less disturbing, than using for example line markers, as shown in [56], among other applications. Note, finally, that the visible ranges can be both smaller, but also larger, than the actual ranges. $M1$ and $M2$ in Fig. 5.4 are an example of the latter.

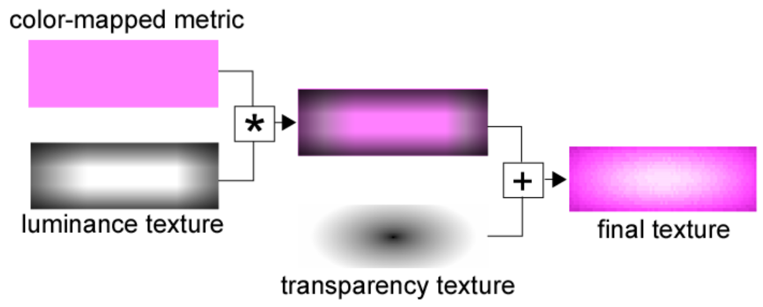


Figure 5.5: Texture design for the metric legend

Grouping metrics

In practice, different software metrics may have completely unrelated meanings and ranges. For example, it may not make sense to compare a lines-of-code metric with a safety metric. Conversely, there are cases when we do want to compare two logically related metrics, *e.g.* lines-of-code and lines-of-comments. We support both scenarios allowing related metrics to be *grouped*. Grouped metrics are marked by being surrounded by a black frame and a superimposed translucent gray cushion, both in the metric legend and the metric lens, see for example $M1$ and $M2$ in Fig. 5.2 left and right, respectively. Several groups can exist in a single visualization, the meaning thereof being that the user should compare the ranges and values of only those metrics located in the same group.

All in all, the metric legend is a compact way for both specifying and understanding the metric lens values in the UML diagram view. A typical usage scenario for the two widgets would proceed as follows. First, one selects which metrics to visualize from a potentially large precomputed set, by enabling their checkboxes in the metric legend. Second, those metrics which are related, from the perspective of the analysis at hand, are grouped in the metric legend. Third, the visual ranges of the selected metrics are adjusted, *e.g.* to focus the analysis on a specific sub-range where most values of interest lie. Finally, the resulting metric lens visualization is examined to discover important metric-metric and metric-structure correlations, by looking at the metric icon colors and bar lengths to spot value outliers, distributions of metric values within each class, and distributions of values across different classes. Tooltips in both the metric lens and legend indicate the actual metric values under the mouse. Colors in the metric lens are interpreted by using the metric legend.

5.4.3 Implementation details

We implemented the metric lens-and-legend combination described so far in a fully integrated reverse-engineering tool aimed at C++ code bases. As outlined in Section 5.3, the metrics-and-structure data mining is done using third-party C++ parsers. The metric lens and metric legend visualizations described in Section 5.4.1 and 5.4.2 are implemented using OpenGL. Specifically, the metric lens technique is used to draw the visible metrics atop of each class icon, scaled and colored as indicated by the metric legend.

Since we extract UML diagrams directly from source code, we must also provide a layout engine for them. As a basic layout engine, we use GraphViz's *dot* engine[5], which works well on connected directed acyclic graphs (DAGs), such as delivered by classes and their inheritance relationships. *dot* works best for moderate graph sizes, *e.g.* under a hundred nodes. This matches well the size of a typical UML diagram. Before running *dot*, we first compute the class frame heights based on the number of their public class members, and class frame widths, based on the method signature lengths. Class member signatures are available from the architecture extraction phase (Section 5.3.1). Next, we lay out the diagrams using *dot*, considering only the inheritance relations. Finally, we draw the resulting graphs, adding the association relations too. This delivers a quick, but robust, layout method, with predictable results. If desired, more sophisticated layouts can be substituted, *e.g.* as provided by the GDT [32, 21], SugiBib [25, 26], Tom Sawyer Software [111], or VCG [52] tools, at the expense of a more complex implementation.

Additionally, we visualize other architectural aspects using the *areas-of-interest* (AOI) visualization technique (Chapter 3). Sets of classes in a diagram which constitute an aspect, *e.g.* all classes involved in a given design pattern or functionality, are drawn surrounded by the AOI contour (see *e.g.* Figure 5.6).

5.5 Case study

To assess the effectiveness of our proposed combined lens-and-legend metrics visualization, we conducted a case study. The study's goal was to get an understanding of the modularity and maintainability of a given C++ code base containing more than 15000 lines of code. The system, a UML editor, has been developed by several individuals over a period of four years. The editor has been developed using the MetricView tool [107] as a basis and contains all the functionality described in Chapters 3, 5 and 6. The current developer, involved in the last few development years, has mentioned the existence of several maintenance and evolution problems, related to the use of different coding styles, badly documented parts, and undocumented dependencies between the various classes in this system.

Our study's main question was: Would a outside person (the investigator), who is *not* involved in the system's development but is experienced in C++ development and reverse-engineering in general, be able to use our visual software analytics tool for a short period of time, and derive insight regarding the maintenance problems of the system under scrutiny, which would be confirmed by the current system developer?

To answer this question, a UML model, several software metrics, and areas-of-interest denoting smaller subsystems, were first extracted from the code base by the current developer, as described in Section 5.3.1. As relations, we consider inheritance and association, the latter being further specialized to data object usage and function call (that is, an entity *A* is associated to an entity *B* if it uses a data object from within *B* or calls a function or method from within *B*). Containment relations, such as functions and data members in classes, are also extracted. These form the basis of the UML class icon construction in the visualization.

We did not involve the investigator in this task, so that additional insight derived during the setup of the extraction phase should not bias the actual visualization evaluation. The investigator was then given access to the lens-and-legend visualization, and told which metrics were available in the database, as well as given a very brief (under 10 minutes) description of the functional aspects captured in each UML diagram. The investigator next performed three types of analyses: a complexity assessment (Section 5.5.1), a change propagation analysis (Section 5.5.2), and a code-level documentation review (Section 5.5.3). All analyses took under two hours, and involved chiefly the usage of our visualization tool. The actual C++ source code of the analyzed system was investigated only for about 10-15 minutes, to check some hypotheses which were not evident from the visualization alone. Finally, the investigator reported and cross-checked his findings in a discussion with the current developer.

The analyses performed in this case study are described next.

5.5.1 Complexity Assesment

In this first analysis, we aim to understand how complexity is spread over the system's structure, in search for so-called complexity hot-spots, *i.e.* parts of the system which may prove hard to understand or maintain.

Figure 5.6 shows one of the extracted UML diagrams displaying three areas-of-interest for three subsystems: UML Data Model, Visualization Data Model and Visualization Core (implementation). As relevant metrics for the complexity analysis, the lines-of-code (*LOC*) and McCabe's cyclomatic complexity (MVG) of each method were chosen, using the metric legend widget (shown lower-right in the same figure). The *LOC* metric is visualized using rainbow-colormapped constant-size bars (the left bar-graph in the classes in Figure 5.6). The MVG metric is visualized with purple bars scaled to the metric value (the right bar-graph in the classes in Figure 5.6). Next, the metric lens display was sorted decreasingly on *LOC* (from top to bottom of the class icons), and also the visible ranges of *LOC* and MVG were set to 50 and 10, respectively. This enables one to quickly discover methods larger than 50 LOC and/or having a complexity above 10, which are figures that was considered by the user to indicate a "complex" method, simply by looking for red, respectively long purple, bars in Figure 5.6).

Looking at Figure 5.6, we quickly see that the most complex and large classes (by both number-of-methods and methods LOC) belong to the visualization subsystem. Although there is no strict correlation between complexity and size, we still see that the Data Model classes are quite small and of low complexity. Brushing the method names with the mouse, we see indeed that most of them are *get()* and *set()* data-accessors, which are indeed simple and short. We conclude that the Data Model subsystem is relatively simple and easy to maintain.

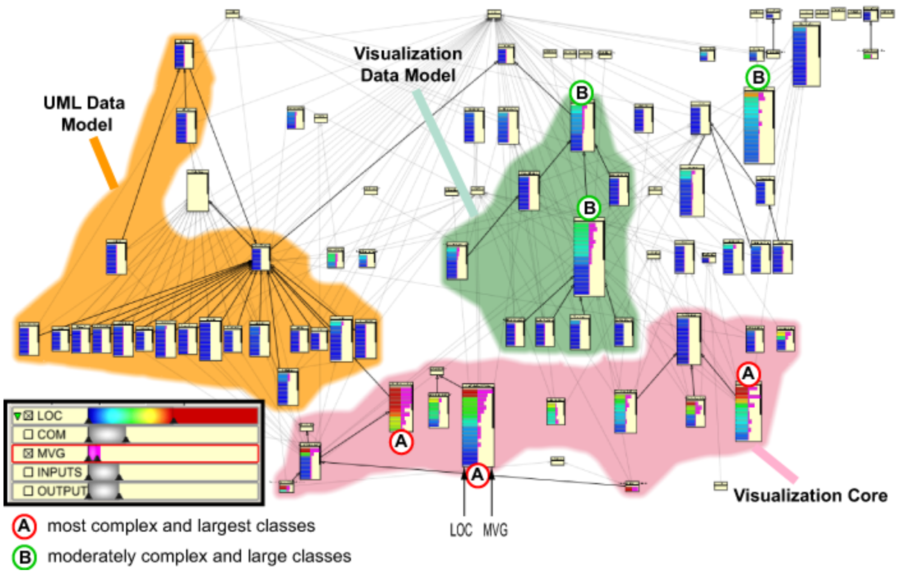


Figure 5.6: Complexity assessment of a UML diagram with three subsystems.

In contrast, the Visualization Core subsystem contains quite large classes, having quite large methods (warm colors in left bar braph), and also the largest and most complex classes (marked A in Figure 5.6). This subsystem concentrates likely the highest complexity. Finally, the Visualization Data Model subsystem contains quite small and low-to-medium complexity classes (*e.g.* the two marked *B* in Figure 5.6).

Overall, the conclusion is that complexity (and size) are not uniformly spread over the system architecture. The Visualization subsystem contains the largest and most complex classes, while the other two subsystems contain relatively simpler and smaller classes, with only a very few moderately large and complex classes, too. As such, maintenance (and development) effort seem clearly to be focused on the Visualization subsystem.

5.5.2 Change Propagation Resilience

In the second analysis, we would like to assess if our system is resilient to changes. In other words: would a change in the code of a class trigger lots of changes in other classes, due to data-dependencies? Spotting such situations is essential to assess the maintainability of a system, as many cascading changes indicate a hard-to-maintain system.

We use the same UML diagram as for the complexity assessment, but now consider the number of variables read ($INPUTS$), respectively written ($OUTPUTS$), by each method. Metrics are sorted in decreasing order of $INPUTS$, and visualized with scaled bars, blue for $INPUTS$ and purple for $OUTPUTS$. Both ranges of $INPUTS$ and $OUTPUTS$ are set to the same value, since the metrics have the same dimensionality. The resulting visualization is shown in Figure 5.7.

We quickly see that there is no evident correlation between $INPUTS$ and $OUTPUTS$

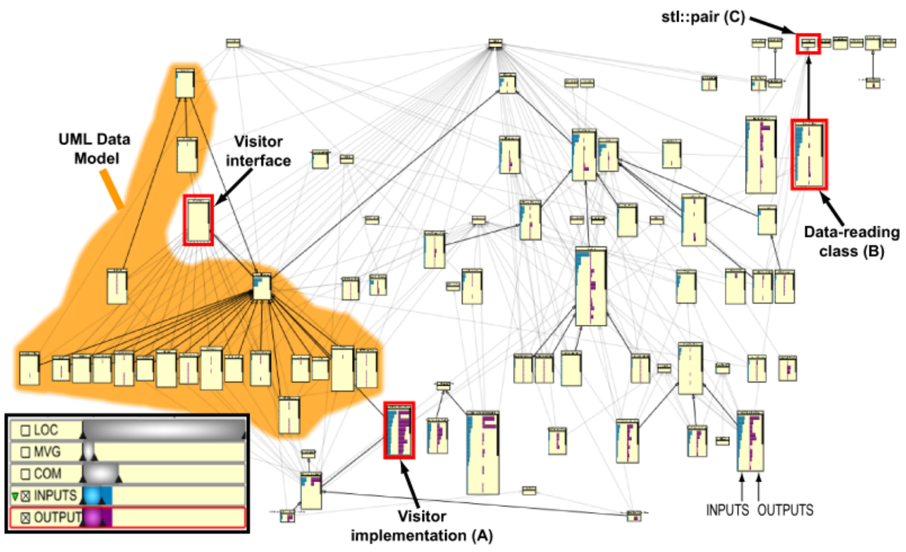


Figure 5.7: Change propagation analysis.

values, but also discover some interesting outliers. The class marked *A* reads and writes a lot. This class, called *UMLModelVisualizer*, is responsible for the rendering, or visualizing, of UML model elements. Following the UML diagram, we discover it inherits indirectly from a *Visitor* interface (shown marked in red in Figure 5.7). Looking at the method signatures in class *A*, we understand that it accepts objects of UML Data Model types through its *Visitor* interface. A quick code browse of this class shows that the high read and write metrics are actually due to the *Visitor* pattern implementation. Since this is a clean design pattern, it was assessed that the strong dependency of *UMLModelVisualizer* from the Data Model subsystem is a safe, acceptable one.

Another outlier class, marked *B* in Figure 5.7, reads a lot of data (high *INPUTS* metrics on most of its methods). Looking at its association relations (arrows on the UML diagram), we discover that this class has a *single* relation, which is actually an arrow (read) pointing to the *std::pair* class, which belongs to the Standard Template Library (STL) C++ library. Since STL can be considered as a very stable component, whose interfaces should only rarely change (if ever), we conclude that our class *B* is also resilient to change.

Now, we add to the *LOC* metric to our analysis, to discover whether read or write-intensive methods are also large ones. Such a case would be undesirable, since it would imply that methods that 'couple' their class with other classes, by reading or writing data, are also intricate methods, which are difficult to change when adding or removing a data dependency. Ideally, the methods that couple different classes should be simple, so modifying the data dependencies should cause only limited code modifications.

Figure 5.8 shows a zoom-in on the same UML diagram as in Figures 5.6 and 5.7, this time showing the *INPUT* and *OUTPUT* metrics grouped (since we want to visually compare them on the same scale), drawn with scaled blue and purple bars respec-

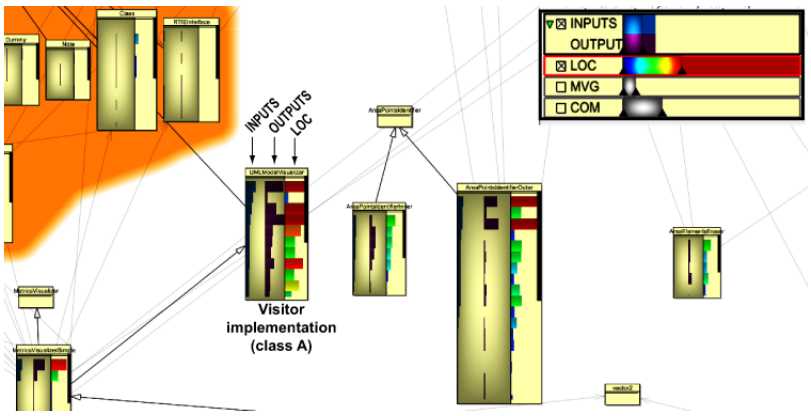


Figure 5.8: Correlation of number of reads and writes and method sizes

tively, and the *LOC* metric drawn with scaled, rainbow-colored bars. The grouping of the *INPUT* and *OUTPUT* metrics is visible in the shaded texture covering the left and middle metric-lens columns displayed within the classes. The third metric, *LOC*, drawn as the rightmost column in the metric lens, is not contained under this texture, meaning that it is not grouped with the former two. If we now look at the same *Visitor* implementation class *A*, which was identified as having the largest number of reads and writes connecting it to other classes, we see that it writes more data than reads (*OUTPUT* metric larger than the *INPUT* metric for most methods). Additionally, we see that methods writing the most are also its largest methods - the *OUTPUT* metric is high when the *LOC* metric is also high. Given the purpose of this class, it is expected that these are the methods where the core of the UML rendering activity is concentrated. A detailed code investigation showed that this is indeed the case.

Overall, the relatively low values of the *INPUT* and *OUTPUT* metrics indicate that code changes due to data access changes should not be large, so the system should not require a lot of recoding in the case its architects decide to change the way data is accessed. The largest number of data accesses occur in the *Visitor* implementation (class *A*). However, as we can also see on the diagrams, this class is actually connected with *only* another class, namely its superclass, via inheritance. Hence, all its data accesses, albeit numerous, are localized: they go to the superclass, probably as a part of implementing the *Visitor* interface. This is a good sign: if this class ever needs to be changed due to data access changes, its changes are limited to its interaction with its superclass.

5.5.3 Code-Level Documentation Review

In our third study, we would like to assess which parts of the code are well commented (or not), and in particular how this happens for the largest methods. Having a few uncommented system components is not critical for maintenance, but having a system where the most complex or tightly-connected components are poorly commented is a typical sign of unmaintainable code.

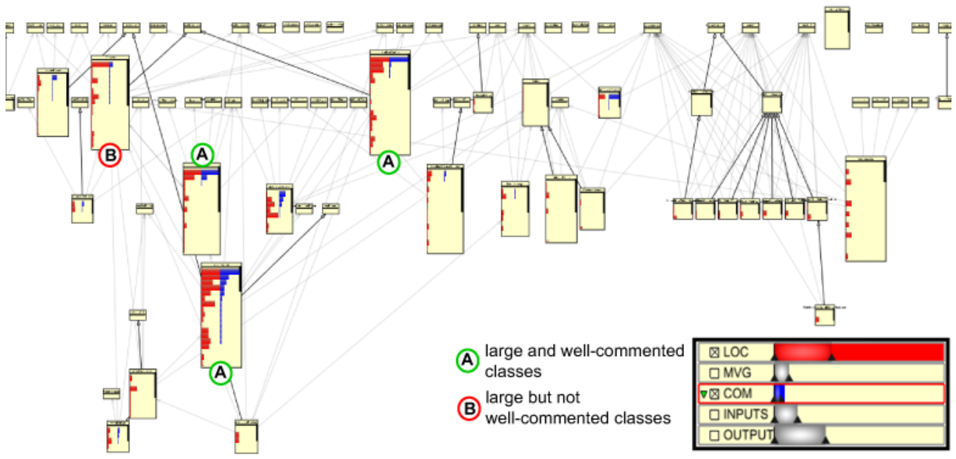

A
B
large and well-commented classes
large but not well-commented classes
LOC
MVG
COM
INPUTS
OUTPUT

Figure 5.9: Code-level documentation review

For variation, we consider here another diagram which represents the Graphical User Interface (GUI) of the system and the visualization part related to GUI. We display the *LOC* and comment-lines (*COM*) metrics, sorted in decreasing order of *COM*, using red, respectively blue scaled bars (Figure 5.9). This emphasizes the best commented parts. Using the metric legend, the visible maximal values of *COM* and *LOC* are set to be in a ratio of 1 to 7. This means that equal-length metric bars in the visualization will indicate methods having one comment-line (or more) per 7 code-lines, which was considered to be a good comment-to-code ratio.

Looking at Figure 5.9, we see that in most cases there are no blue bars of equal length to the red bars, indicating that most methods are not well commented. However, for the largest methods, such as those at the top of the classes marked with *A*, this situation is better: here the blue and red bars are roughly equal, indicating a good comment-to-code ratio. We also discover a class (marked *B*) which has quite many methods, and contains one of the largest methods (red bar at top) with very little comments. Overall, this seems to be one of the weakest classes from a documentation perspective: it has many methods, some of which are large, and those are badly commented. Moreover, this class is also connected with several other classes: it inherits from a STL container (shown by the bold arrow connecting it with the small class eight from left on the top row in the figure), but also has 12 other associations with 12 other classes (shown by the light gray lines connected to this class). We have here, thus, a quite large class, with many methods and connections to other classes, but which is badly commented. This is a good candidate for adding documentation.

5.6 Discussion

During the validation phase, the developer confirmed the observations made, and conclusions drawn, by the investigator. Besides confirmation, he also exposed the reasons for which the various outliers detected by the visualizations, *e.g.* large classes concentrating high complexity; large and well-documented classes; and classes tightly coupled in various design patterns, occured in the design. The Data Model and Visitor pattern parts are cleaner than the Visualization Core part, that includes the largest and most complex classes, which are currently in an unstable state and contain mostly highly experimental code. Overall, the considered system can be quite clearly split into a stable, clean, maintainable 'legacy' part, and a lower-quality, complex part containing code still under development. Of course, these reasons could not have been deduced solely by using the visualization and without some understanding of the functionality of the system under study. However, we argue that our study succeeded in the sense that a programmer with no knowledge of the studied system could locate quite reliably a number of design patterns and maintenance-related bottlenecks in the system structure, using chiefly the proposed visualization, in a very short time.

The typical operations involved in setting up analysis scenarios with our proposed techniques are quite simple: select a number of metrics of interest; tune the visible ranges to reflect ratios or maxima that one wants to check (or one expects) in the code base; and sort on the different metrics to see metric distributions and detect their correlations over all methods. Overall, constructing visualizations like the ones presented here can be truly done with just a few clicks in the metric legend. This assists users to actively explore large systems with many diagrams and metrics, by massively diminishing the amount of time needed to check a correlation or distribution of some metrics over some subsystem.

Combining the metric icons with the subsystem partitioning rendered as areas-of-interest (AOIs) effectively helped us understanding the relations between metrics and structure at a finer level than diagrams themselves. As such, the typical analysis scenario observed was: loading several UML diagrams, toggling through their visualizations, then concentrating on a particular metric or metric-structure pattern, possibly in conjunction with a given AOI, and finaly zooming in at class-level (Sec. 5.4.1) to read the methods' names.

A recurring question is how our proposed techniques compare to related work in visualizing structure and metrics, *e.g.* in the CodeCrawler, sv3D, and CodeCity systems discussed in Chapter 2. We see several differences, as follows.

First and foremost, our underlying visualization uses a UML diagram layout and look-and-feel, in line with our first requirement stated in Chapter 1. Although other visualizations, such as the ones named above, can produce more compact layouts and thereby are visually more scalable, our observation is that UML diagrams are easier perceived and understood, and thus more accepted, by typical software engineers.

Secondly, the way we add *method*-level metrics to our UML diagrams is different. In the above systems, methods are rendered as separate geometric entities, *e.g.* squares or 3D bars, and method-level metrics (when visualized), are mapped to geometric attributes of these entities, such as width or height. This creates treemap-like visualizations, in which methods are sorted in a two-dimensional layout. These have a different flavor and pose different interpretation challenges as compared to our table lens visualizations, where methods are sorted in a one-dimensional layout (the table). Showing several metrics in the above visualizations means mapping them to *different* visual attributes, such as size and color. In contrast, we map metrics strictly to the table lens icons, which are all rendered in the *same* way, and not change the layout or size of the methods themselves, which are rendered as text in the classical UML diagram style. Given our uniform visual mapping of metrics to the same type (and size) of metric icons over an entire UML diagram, we argue that metric-metric comparisons are easier to achieve in our case. Additionally, the diagram layout is always *fixed* in our case, whereas the other named methods typically change the layout to accommodate variations in the method icon sizes given by the metric-to-size mapping. Layout changes are well-known to be disruptive, especially when users have learnt a given system and expect to find the same entity in the same place, and having the same size, on a given diagram.

A third difference resides in the use of the third dimension. The sv3D and CodeCity systems discussed in Chapter 2 use icon height to map one metric value. Although this lets users quickly spot maximal outliers as high icons, this also brings several well-known undesired aspects of 3D software visualizations, such as occlusion and the difficulty to choose a good view point.

Finally, we provide explicit support in constructing (and interpreting) visualizations that target specific questions, by the functionality of the metric legend widget. As far as we are aware of, the other named visualizations do provide options to customize the visualization in terms of configuration scripts or similar mechanisms, but do not encode these explicitly and compactly in a visual interactive representation. As such, we argue that our method better supports the quick design visualization that answer questions posed on-the-fly during system understanding, with a minimal amount of learning. The fact that both our table lens and metric legend are two-dimensional, tabular, visualizations makes their visual correlation easier than, for example, in the case one would like to add a similar metric legend widget to software visualizations which encode metrics in a different way and lay out metric icons differently.

Our focus here is on visualizing the combined structure and metric data, and not on extracting it. However, the presented metrics-and-structure visualization can be quite easily integrated with other reverse-engineering pipelines involving different code analyzers for C++ and/or other object-oriented languages, *e.g.* Java or C#.

Finally, a word on visual scalability. By controlling the diagram creation, we limit the number of elements per diagram and allow for several diagrams, hence making the layout of a single diagram scalable. The display of several tens (or more) of methods per class is highly scalable, given the underlying table lens technique (Section 5.4.1). The strongest scalability limitation regards the number of metrics that can be shown in the same time on a class. In our experiments, we saw that displaying more than three different metrics per method on even a part of a diagram makes the result overcrowded and hard to read (see *e.g.* Figure 5.10). Increasing the class frame widths alleviates this problem, but can produce too wide diagrams which are unnatural or occupy too much space. However, it can be argued that this is not a major problem for the typical use cases of correlating metrics with structure in examining software system diagrams. True, in such scenarios, one may need to examine many metrics, but we are not aware of typical scenarios in which the number of metrics that are to be examined *simultaneously* in a single scenario would exceed a few (2..4).

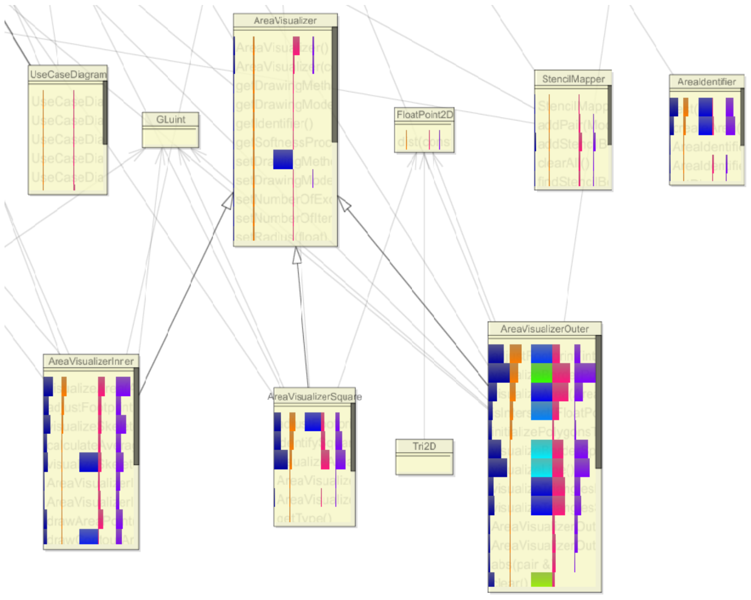


Figure 5.10: Class frames with five metrics

5.7 Conclusion

In this chapter, we have presented a technique to support visual analyses related to member-level metrics on software diagrams. The method constitutes of two parts: a *metric lens*, which is essentially a table lens, adds metric information within the typically limited space of a diagram element; *metric widget* helps both in tghe specification and interpretation of the metric lens visualizations. Overall, the presented techniques work well for tasks such as spotting metric distribution and metric value outliers and help in the task of correlating such outliers among themselves and with the system structure. These tasks are basic ingredients of more complex activities in software understanding involving architectures and metrics. Concerning the visual design, a desirable fact is that we allow a certain scalability of the visualization without having to change the layout of a given diagram. However, this constraint also ultimately limits the method's scalability to approximately three metrics per element for typical layouts of UML class diagrams.

Chapter 6

AOI-level metrics

In this chapter, we present a new method for the combined visualization of software architecture diagrams, such as UML class diagrams or component diagrams, and software metrics defined on groups of diagram elements. To this end, we extend the rendering technique for areas of interest presented in Chapter 3 to visualize several metrics, possibly having missing values, defined on overlapping areas of interest. We use a solution that combines color mapping, texturing, blending, and smooth scattered-data point interpolation. The presented method simplifies the task of visually correlating the distribution and outlier values of a multivariate metric dataset with a system's structure. We demonstrate the application of our method on component and class diagrams.

6.1 Introduction

As outlined in Chapter 2, software metrics can be defined on software architecture diagrams at several levels of detail. For object-oriented systems, these levels are class member, class, and group of classes. In Chapter 5, we have presented a method that uses a modified table lens metaphor to visualize member-level metrics on UML diagrams. The visualization of class-level metrics on UML diagrams is covered by previous work, notably [107, 48]. These visualizations aim at enabling the users to correlate metrics with each other and with the system architecture, and therefore require effective ways to combine the presentation of several metrics and the architecture in a single picture.

In this chapter, we consider the last type of metrics, which are defined on *groups* of software components (*e.g.* classes), also called areas of interest (AOIs). We extend the technique for rendering AOIs (Chapter 3) by using color mapping, blending and texturing to render several metrics, defined as a multivariate dataset with potentially missing values, atop of such areas, so that users can spot metric-metric and metric-area correlations. Our technique removes the need of drawing metric icons atop of the diagram elements, as it is done by related visualization techniques [107, 48]. Hence, we can use the space within the diagram elements to show other information, such as class member metrics, using the metric lens technique (Chapter 5), or text annotations.

This chapter is structured as follows. Section 6.2 details the analysis tasks and the related visualization requirements and challenges for AOI-level metrics. Section 6.3 presents our new technique for rendering several metrics atop of AOIs. Section 6.4 presents case studies of using our new rendering technique on UML and component diagrams. Section 6.5 discusses the obtained results (Sec. 6.5.1) and using different color schemes for visualizing metric values (Sec. 6.5.2). Section 6.6 concludes the paper.

6.2 Visualization requirements

Let us first introduce the data model for our AOI-metrics visualization. This data model extends the data model presented in Section 3.2 by adding area-level metric information.

Consider a system 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. 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 [123]. If desired, several metrics m_i^k, $k > 1$ can be defined over a given area of interest A_i, in the same way as above. In the following, we shall consider only one metric per area of interest, for the sake of exposition simplicity.

AOI-level metrics occur in many software analysis applications when

- a given software metric is defined *only* over a subset of the elements of a given diagram;
- these elements share some common property, or aspect.

For example, imagine a software system diagram in which a subset of the elements are involved in multithreading and another, possibly overlapping, subset in involved in performance-critical operations (a similar concrete example will be presented later in Section 6.4.1). First, if we want to visualize the elements involved in the two system aspects, we could define two areas of interest A_{thread} and A_{perf} and render them as described in Chapter 3. Now assume that we compute two metrics on the elements involved in the above areas, namely m_{safety} that gives a safety measure of all multithreaded elements in A_{thread} and m_{speed} that gives the speed of all performance-critical elements in A_{perf}, respectively.

The question is, now, how to support tasks such as correlating values of these metrics with the AOIs *and* the overall system structure. This implies being able to perform several subtasks, such as:

- correlate the elements with the AOIs and overall system structure,
- correlate the elements with their AOI-level metric values,
- correlate several AOI-level metrics among themselves.

The first subtask is already addressed by the AOI visualization method presented in Chapter 3, which draws AOIs atop of a given system diagram, and enables users to tell which elements are in a given AOI and also which AOIs contain a given element, *e.g.* which are all multithreaded elements and whether a given element is multithreaded and/or performance-critical, in our example.

The next two subtasks relate to the question of how to show the AOI-level metrics. Let us discuss this aspect in more detail. The set of metrics m_i defined over all areas A_i in a given diagram could be seen as a multivariate scattered-point dataset [90], with elements j as data points and the metric values i as variables. Hence, if we view the metrics from an element-centric position, that is from the perspective of each element j, we could say that our task is nothing more than showing a set of metric values m_{ij} at a given element j. We could approach this problem in the same way as we solved the task of visualizing method-level metrics, *i.e.* using a table lens with i values m_{ij} rendered atop of element j (Chapter 5), or alternatively using metric icons rendered atop of the same element j as described in [107, 48]. This solution would allow the comparison of all metric values for a given element. However, there are several drawbacks.

Consider, for example, the method of Termeer *et al.* that shows element metrics using icons scaled and colored by metric values, drawn atop of the elements [107]. Although quite intuitive, this approach has several drawbacks. Let us examine a diagram with five metrics defined over five areas of interest (Figure 6.6), visualized with metric icons. First, we see that icon sizes are constrained by the element sizes, which can be quite small, since one of our main requirements mentioned in Chapter 1 was that the layout of a diagram, including the sizes of its elements, should be fully at the discretion of the user. If elements are small, it is hard to see specific metric values, since their icons will be also quite small. Second, we want to keep the element surfaces free to draw other data, such as method names and annotations. Hence, we cannot draw area-level metrics within the elements. 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 metrics, rendered within elements with icons, to areas, rendered outside elements with textured contours (Chapter 3). In other words, we want to present the AOI-level metrics from an area-centric, rather then element-centric, perspective, *i.e.* emphasize the relationship between these metrics and the areas they are defined on. Similar problems arise when using the table lens technique. A second example of these types of problems is presented later in Section 6.4.1.

There is a related problem related to the possible use of icons or table lens techniques to show AOI-level metrics. if we want to visualize both AOI-level and class-level or member-level metrics on the same diagram, we cannot use icons or the table lens technique for AOI-level metrics , since the space within the elements is already occupied by icons showing the latter two metrics.

Considering the above, the specific refined requirements for an AOI-level metric visualization are as follows:

- visualize AOI-level metrics so that metric-metric and metric-AOI correlations are easy to perform,
- keep the space within the elements free for visualizing class-level or method-level metrics.

Besides these requirements specific to our current goal of showing AOI-level metrics, we want also to satisfy the overall requirements of UML-like aspect, understandability and scalability introduced in Section 1.5.

6.3 Texture-based solution

If we summarize the requirements of AOI-level metric visualization discussed in Section 6.2, we state that 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,
- we do not draw metrics on the elements themselves.

The AOI drawing method, presented in Sec. 3.6.2, constructs a contour that encloses the elements located in an area of interest (see *e.g.* Figure 3.21). We show next how to render several metrics so that metric values, elements, and areas of interest can be easily correlated, and the space within elements is left free for visualizing annotations or class-level or method-level metrics. We use a two step solution. First, we render the values of a single metric m_i over a given area A_i (Sec. 6.3.1). Next we combine all metrics m_i for all areas A_i in a single image (Sec. 6.3.2). Finally, we add shading to the areas to further emphasize their structure (Sec. 6.3.3). In the implementation of our proposed solutions, texturing and blending will play an important role.

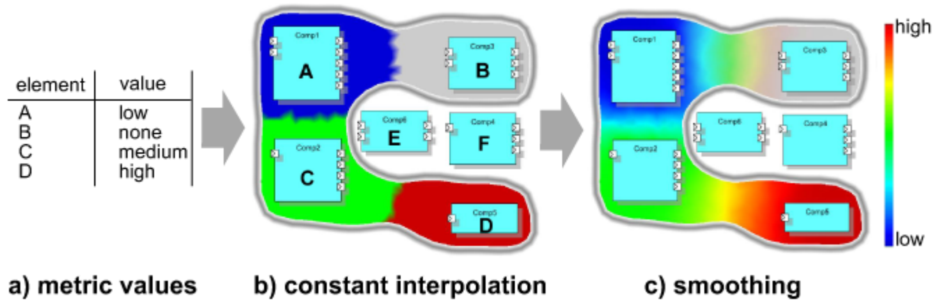
6.3.1 Rendering a single metric

They key idea used to address the requirements outlined in the previous section is to render 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, for the moment, 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 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 [85]. 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



element	value
A	low
B	none
C	medium
D	high

Figure 6.1: Smooth interpolation of element metrics over an area-of-interest

of $\{m_i\}$ over A. Figure 6.1 b shows $\mathscr{M}$ for the metric values in Figure 6.1 a, using a red-to-blue colormap (or any other colormap, as desired). 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

$$h(x) = rainbow(\mathscr{M}(x)) \tag{6.1}$$

$$s(x) = \mathscr{P}(x) \tag{6.2}$$

$$v(x) = 1 \tag{6.3}$$

where *rainbow*() is the chosen colormap (see Figure 6.1 right). 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, as described in Chapter 3.

In the final step, we smooth our piecewise-constant interpolation. For this, we apply a Laplacian filter [19] 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 Figure 6.1 c). The idea behind this smoothing step is that it is easier to visually follow smooth color changes across an AOI than the piecewise-constant color variation, as the user does not get distracted by the sharp color changes occurring at the edges of the underlying Voronoi diagram. This point, and the implications of using smooth interpolation on metrics defined as a scattered point set, is detailed later on in this chapter in Section 6.5.2.

6.3.2 Combining several metrics

Now we must combine several metrics defined on possibly overlapping areas, each of them rendered as described in the previous section. We cannot simply additively blend areas of different colors as described in Chapter 3 (see *e.g.* Figure 3.25), as this would mix the individual colors which show metric values beyond recognition, even when only two areas overlap.

We use a texture-based solution, as follows. To each area A_i, we assign a different texture. We carefully designed a small set of textures (see Figure 6.2). The overlap of any textures in this set is intended to create a visually different pattern. The textures contain just opacity data: black denotes opaque zones, white gaps are fully transparent, gray indicates an alpha value between 0 and 1.

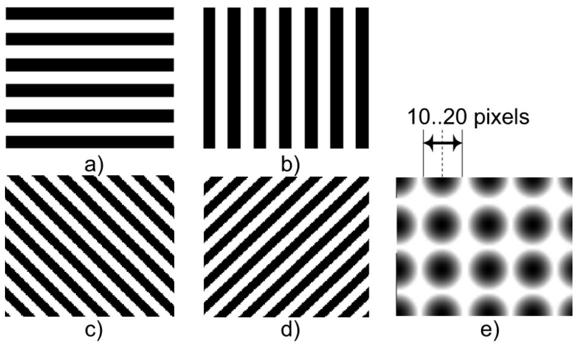


Figure 6.2: The proposed set of textures. Gray value denotes opacity

We now render each area A_i by combining its color (showing metrics) computed by interpolation (Section 6.3.1) with its transparency texture (showing the area's identity) using OpenGL's texture modulation capability. Figure 6.3 shows the application of texture *c* from Figure 6.2 on the area. When we have several areas, we draw them starting from the largest to the smallest one, so that small areas appear atop large areas, as this maximizes information visibility.

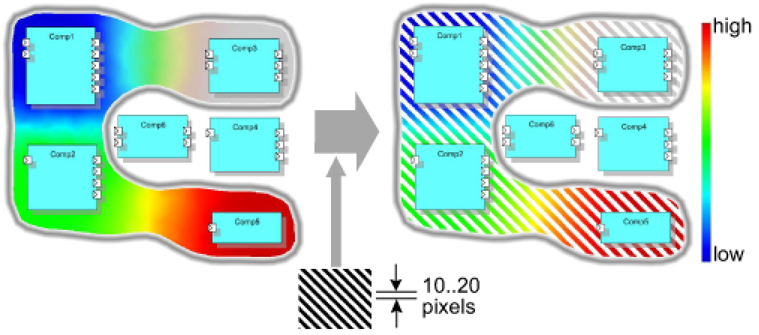
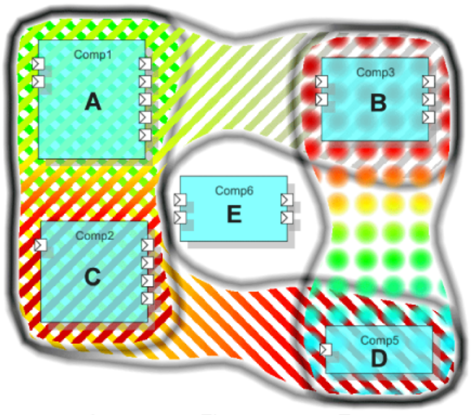


Figure 6.3: Applying a texture on the area

Figure 6.4 shows the application of the texturing idea to 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.



Area	Elements	Texture
A_1	B, D	●
A_2	A, B, C	▨
A_3	A, C, D	▧

Figure 6.4: Diagram showing three areas of interest with metrics

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 color 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.* Figure 6.12 where only a few iterations are done, which yields well-separated color areas around the elements, and almost no interpolated colors. 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 the method, if these are seen to produce less ambiguous results. More on this issue of color interpolation is presented in Section 6.5.2.

Note that the color smoothing is of additional value in the case of overlapping areas. If several such overlaps exist in a visualization, it would be quite difficult to distinguish how values in each area change at overlap points if we had sharp color transitions caused by *both* the texture stripes and the sharp color borders given by the Voronoi-like color assignment described in Section 6.3.1. Color smoothing eliminates the latter, so sharp color changes are only caused by the texture weaving.

6.3.3 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. 6.3.1), 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 3..7 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. 6.3. This darkens areas close to their borders, but keeps them bright in the middle. Normalization ensures that shading is always bright (*i.e.* equal to one) in the middle of an area and dark (*i.e.* equal to zero) 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.

A cheap way to prevent this is to increase the holes' opacity α in the texture patterns from 0 (fully transparent) to 0.2 (slightly opaque). Overall, this gives the effect of convex, shaded 3D shapes - compare Figure 6.4 (no shading) with Figure 6.5 (with shading). At overlaps, the shaded shapes get woven by blending. The darkened borders help to visually separate areas (see the images in Section 6.4). 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, as if texture colors would 'bleed' into the holes. It may be argued that 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 (for a rendering variant that does not exhibit this color bleeding see further below). 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 for areas textured to show metrics. As a side effect, the textured borders proposed in Chapter 3 used to show the borders of areas can be dropped when using the shading technique.

The simple method to strengthen shading by increasing the texture holes opacity outlined above has, however, the undesired effect of attenuating the texture weaving effect. Consider, for example, two areas A_1 and A_2 rendered with two textures T_1 and T_2, that overlap. Consider a point situated in a hole of, say, T_1 in the overlap region. At this point, we would actually see the color $\alpha * T_1 + (1-\alpha) * T_2$. This is visible in Figure 6.5. This is not precisely what we need: we would simply like to see T_2 through the holes of T_1, just as in Figure 6.4, whereas what we now see in Figure 6.5 is a bleeding effect where the colors showing the interpolated metric values extend to the holes and mix with the colors of the underlying area(s).

We can obtain a texture weaving effect without color bleeding, *i.e.* having fully transparent holes showing the underlying textures and a shaded border, by using a slightly more complex implementation based on pixel shaders. When rendering each area, we replace the blending mechanism offered by the standard OpenGL pipeline by a pixel shader. The shader will set the hue of the current fragment as given by the texture, and the luminance as given by the shading signal $\mathscr{S}$, just as before. In contrast, the shader will set the opacity or alpha value to the luminance signal $\mathscr{S}$ multiplied by the texture opacity, instead of using the constant value $\alpha = 0.2$. In other words, the rendered areas are dark *and* opaque close to the rendered area's border, bright and opaque over the texture stripes far away from the border, and fully transparent in the texture holes far away from the border. The shader that accomplishes this is under 10 lines of code. All shaded images in Section 6.4 are produced using this method. If we look at Figure 6.11, for example, we see the desired effect: holes let the colors of the underlying textures fully show through, and the pixels close to an area border are equally dark, whether in a hole or not.

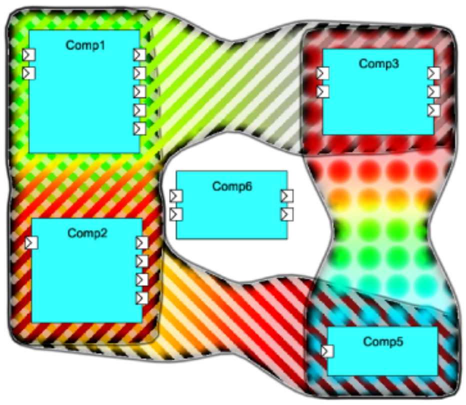


Figure 6.5: Enhanced areas using shading (compare with Figure 6.4)

The overall effect created by our superimposed shaded areas of interest is somewhat similar to the enridged contour maps technique [42]. This technique was used to add a similar shading profile, *i.e.* dark at the borders and bright in the middle, to isolines drawn on weather maps of similar scalar fields. However, there are also some important differences. First, the enridged contour technique assumed that the contours (isolines) are strictly nested, and exploited the shading to communicate the isoline values by means of shading. In our case, shading is used strictly to emphasize the areas' contours, but data values are encoded strictly in the texture hues. For strictly nested areas, *e.g.* the small area containing two elements at the right in Figure 6.11 contained in a larger area, our shading effect conveys a relatively similar effect of having the small area atop of the large area, but this effect is not present when areas intersect. Secondly, the shading is computed differently. The enridged contour maps use a per-pixel computation of the shading based on the difference between the actual value of the contoured signal at that pixel and the value of the nearest lower isoline, whereas in our case the shading is purely based on the geometric distance between the vertex of our area triangulation and the area's border. A further analysis of the similarities and differences of the two techniques could, nevertheless, lead to some interesting enhancements of both methods.

6.4 Case studies

We now illustrate the use of our AOI-level metric visualization in two different case studies.

6.4.1 Analysis of the JPEG decoder performance

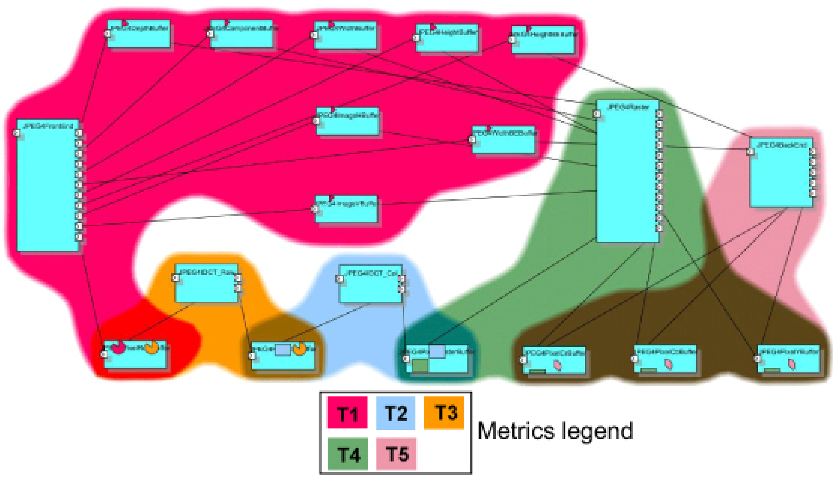


Figure 6.6: JPEG decoder architecture. Icons show the memory usage metric μ_{mem} over five tasks. Areas show the tasks. The metric legend shows the placement of metric icons within each component. Although this figure is quite large, it is hard to correlate metric values and areas

We consider a real-world software project: the architecture of a component-based JPEG decoder [12, 40][1]. The system model was built and its operation numerically simulated using the CARAT toolkit [11]. 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 [12]. We

[1]A more detailed description of the embedding of our visualization techniques in the context of this project is presented separately in Chapter 7

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 system's developers:

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

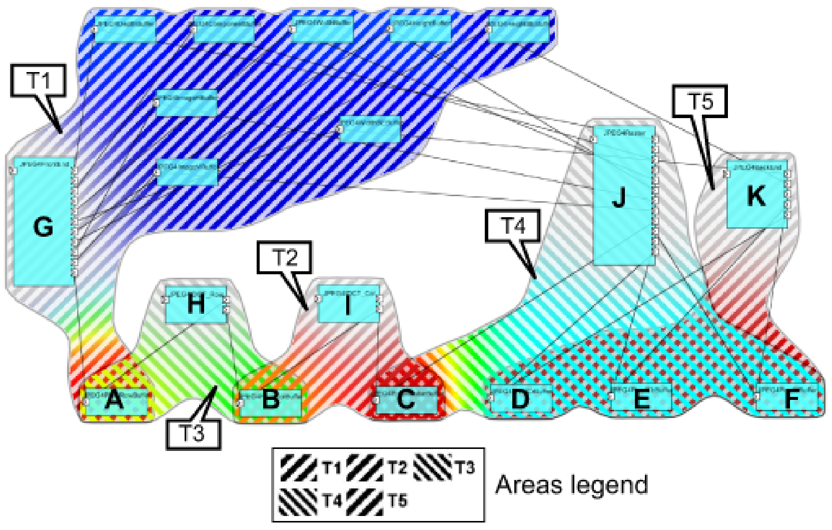


Figure 6.7: JPEG decoder architecture (five tasks with memory usage metric).

To illustrate the advantage of our method, we first use metric icons [107]) 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} (Figure 6.6). The metric legend shows the tasks' colors and also shows where each icon from each task-area is placed within each element (see [107]). However, in Figure 6.6 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 - this is a design constraint ot the original metric icon method. Also, it is hard to visually correlate metric values over large areas. Finally, 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 new technique. Each area (task) uses a different texture (see legend in Figure 6.7). 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, than in Figure 6.6, where the icon-based visualization is used. 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

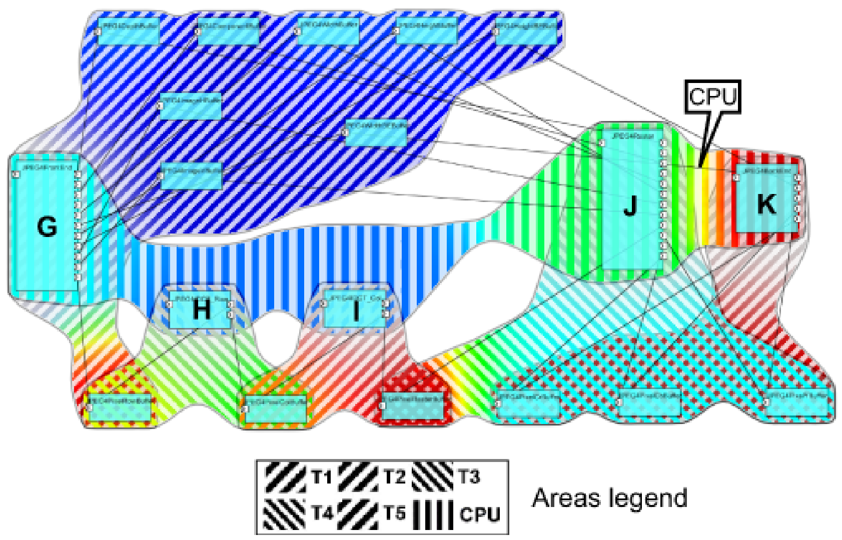


Figure 6.8: JPEG decoder architecture (memory and CPU usage metrics)

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 $D \dots F$ 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 ($T4$).

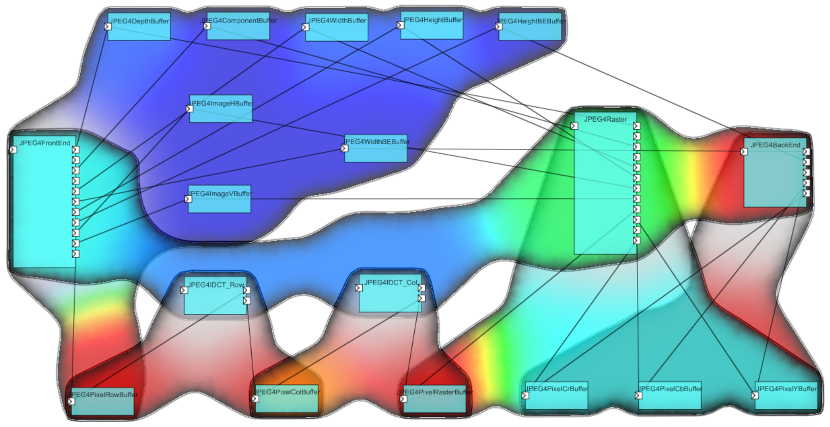


Figure 6.9: JPEG decoder architecture (shaded AOIs)

In our second scenario, we add the CPU utilization metric μ_{CPU} (Figure 6.8). The area A_{CPU}, containing all active components using CPU cycles ($G \dots K$), intersects the task-areas $T1 \dots T5$. To visually segregate the two aspects (tasks and CPU utilization), we use diagonal stripes for the task-areas $T1 \dots T5$ 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 (Figure 6.7). This is correct, as the design of this JPEG decoder splits data (passive) components from algorithm (active) components.

Figure 6.9 shows the effect of adding the shading described in Section 6.3.3. 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 to understand area overlaps. The shaded and textured areas are shown in Figure 6.11

6.4.2 Identifying design aspects during reverse engineering

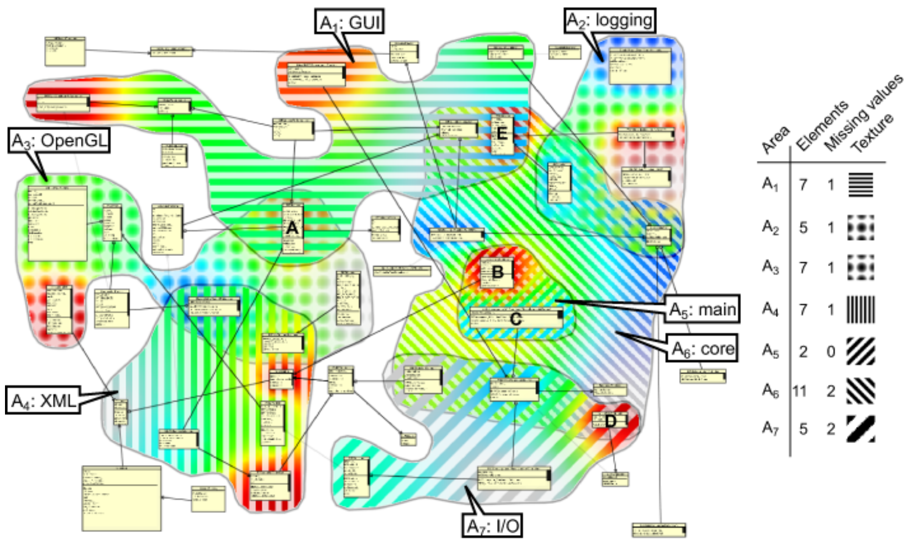
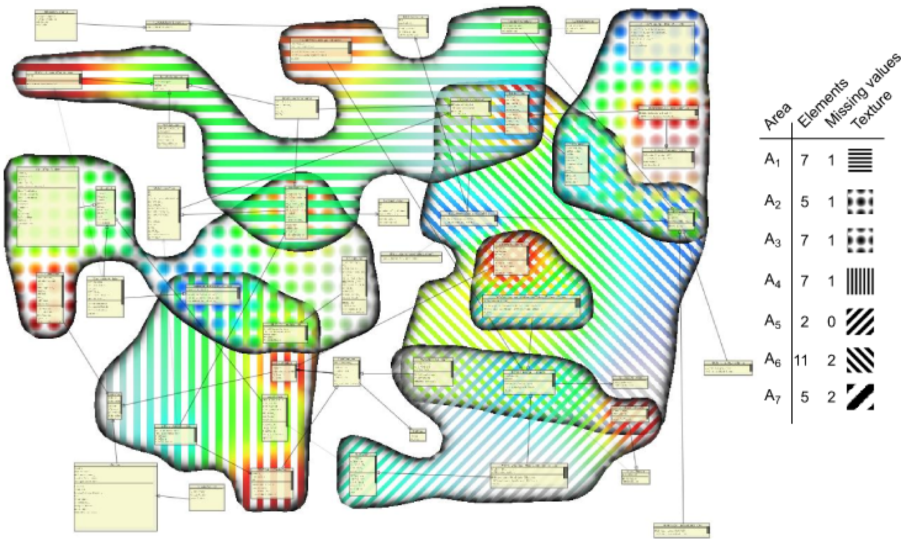


Figure 6.10: Large UML class diagram with 7 areas and over 50 classes. Metrics show the participation of classes in two aspects

In our second application, we extract an UML class diagram from the source code of a C++ graphics editor in a reverse engineering process, using the technique described in Section 5.3. Talking to the system designer, we identified several high-level functional aspects:

- *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. An 'ideal' object-oriented design would require each class strongly involved only in one aspect [51]. 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. Alternatively, other aspect mining techniques could be used, if desired. 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.



Area	Elements	Missing values	Texture
A_1	7	1	≡
A_2	5	1	⁙
A_3	7	1	⁙
A_4	7	1	‖
A_5	2	0	▨
A_6	11	2	▧
A_7	5	2	▨

Figure 6.11: Visualization of the UML diagram in Figure 6.10, now with area shading and half-transparent elements

The entire system is shown in Figure 6.10. 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 6.11 shows the same diagram, areas, and metrics as in Figure 6.10, with shading added. In this figure, shading helps better seeing which elements are in which areas, similarly to the example presented in the previous section.

Figure 6.12 shows a part of another class diagram of the system described in Sec-

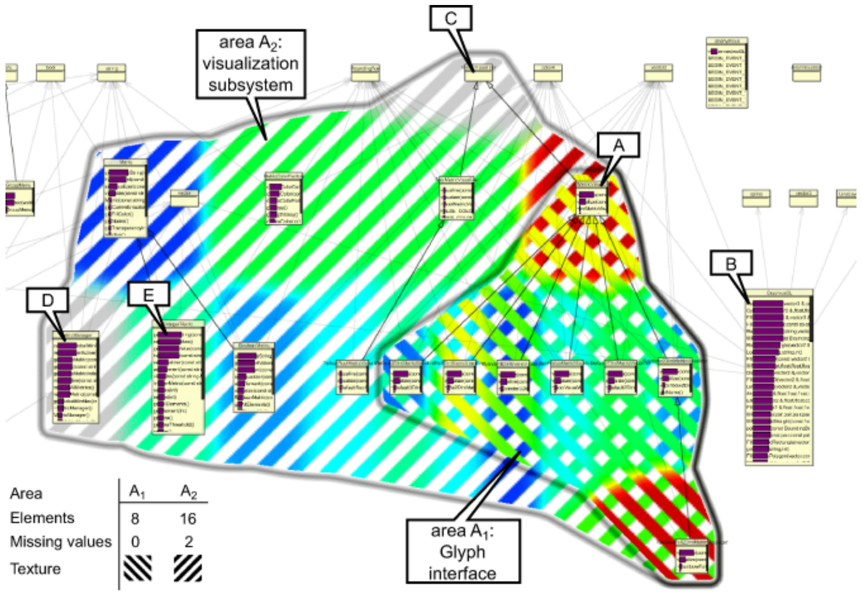


Figure 6.12: UML class diagram with two areas, class-level participation metrics, and method-level lines-of-code metrics.

tion 5.5. 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 AOI-level metric visualization does not draw on classes, we can show an additional metric: the lines-of-code (LOC) for all class methods, drawn atop of classes with purple bars, using the table lens technique described in Chapter 5. 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.

We can use this image 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 in A_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 [51]. Code examination confirmed this: B contains an implementation of different geometric operations. 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 hotspots, 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.

6.5 Discussion

We have conducted several informal evaluation studies of our proposed AOI-level metric visualization technique. Our aim is to compare the effectiveness and acceptance of the new texture-based technique as opposed to the classical icon-based techniques. We specifically compared our new method against [107] as the functionality of both techniques is supported in our UML visualization tool, which offers the same look-and-feel of the UML visualization and GUI controls for both methods. Moreover, we had a relatively large base of users already familiar with this UML visualization tool, in the framework of a 2-year industry-academic cooperation project [40]. The user base includes around 10 professional software engineers involved in creating UML architecture diagrams, such as the JPEG decoder (Section 6.4.1), and computing quality metrics on them.

6.5.1 Results

We discuss several aspects of our technique noticed during the performed evaluation, 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 [85] and Laplacian filter [19] 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 also very fast on any graphics card. The added value of this real-time performance is that users can quickly change all visualization parameters, *e.g.* transparency of the metrics, shading strength, or colormap, and see the results immediately, which encourages exploration.

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 Figure 6.13, rendered with textures shown in the legend. From the 'woven' texture pattern we believe it is possible to distinguish which element is in which area and the colors (metric values) at overlaps. The addition of shading (Section 6.3.3) further helps in separating areas with complex overlaps. Yet, adding a fourth overlapping area can make this image very hard to understand.

However, the question is whether realistic scenarios exist in which one would like to compare or correlate four (or more) metrics in the same time. As mentioned in Section 5.6, we noticed that most software understanding and analysis tasks require the comparison of two, maybe three, such metrics at the same time, but not more. Although we have no hard evidence for the lack of real-world scenarios in which more metrics need to be compared at the same time, we believe that the practical upper limit of three metrics supported by our method is able to cover most practical applications. The same observation, in conjunction with a similar limitation of the number of overlapping textures at one point, was made by Voinea *et al.* [117]. In their method, textures were used to show two or three categorical metrics mapped on rectangles to show software evolution artifacts. In their case, the overlap problem was further aggravated by the small size of the textured elements (a few pixels in each dimension), which practically limited their method to two overlapping metrics in most cases.

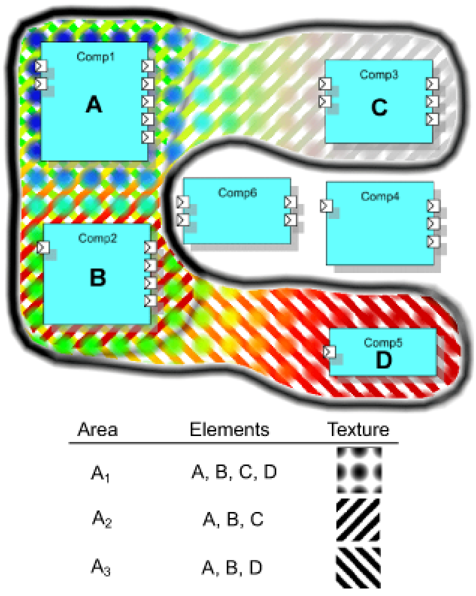


Figure 6.13: Complex intersection of three overlapping areas.

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 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 [117]. 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 (Figure 6.2).

Related methods: To our knowledge, there is only one other software visualization at the time of writing this thesis that uses textures to show numeric metric values [36]. 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.

6.5.2 Color schemes and color interpolation issues

The choice of the color scheme, or colormap, as well as the metric interpolation decision described in Section 6.3.1, deserves a separate discussion.

So far in this chapter, we demonstrated only a simple continuous colormap: the rainbow colormap. Also, we chose to interpolate the metric values (defined on the diagram elements) at the points situated between the elements. The motivation of these choices was the observation that visually following a continuous change in hue, as yielded by a continuous colormap and metric data spatial interpolation (see Figure 6.1 c for example), is easier than visually following a pattern where colors would change rapidly at discrete points (see Figure 6.1 b for example). This observation is well known from the use of continuous colormaps and color interpolation in many application in scientific data visualization.

Yet, a number of issues arise when choosing for continuous colormaps and color smooth spatial interpolation. First, our data values are defined strictly at the locations of the diagram elements. As such, it does not strictly make sense to speak about a data value *between* two diagram elements, as there is no element to carry data there. Secondly, the color interpolation may suggest the existence of data values situated between the values defined on the elements (in data value space). For continuous metrics, such as for example speed or memory consumption, such values have a meaning. For inherently discrete metrics, such as for example indicators that classify a component's performance as "excellent", "very good", "good", "average", or "poor", such in-between values are meaningless, as discussed *e.g.* in [102]. Also, such metrics are typically visualized with discrete colormaps, *e.g.* the diverging colormap shown in Figure 6.14.

From the above observations, we see that there is a tight connection between the nature of the visualized data, the type of colormap used, and the decision whether to interpolate data or not. The colormap and interpolation choices are fundamental to the metric visualization algorithm proposed in this chapter. Given the high impact of these choices on this algorithm, we present next a number of observations related to possible variations in these choices.

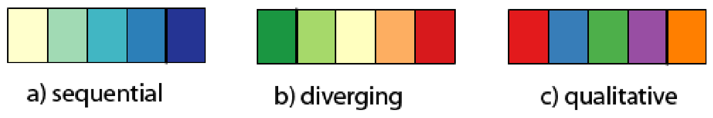


Figure 6.14: Examples of different types of discrete colormaps

The first choice, or degree of freedom, of the AOI-level metric visualization is the actual *colormap* used. Many kinds of colormaps are used in data and information visualization. The main ingredients of a colormap are as follows [81]:

- the type of colormap: continuous or discrete
- the number of colors used
- the actual colors used

Continuous colormaps, sometimes also called color transfer functions, are functions $c : \mathbb{D} \to \mathbb{C}$ that map a data value from a continuous domain $\mathbb{D}$ to a RGB or HSV color space $\mathbb{C}$. In most cases, such colormaps have an infinite number of colors, as for any different $x \in \mathbb{D}$ a different color $c(x) \in \mathbb{C}$ is generated. Discrete colormaps, sometimes also called color tables, are typically used when the data domain $\mathbb{D}$ is a discrete set of elements, like a vector or an enumeration of values. Every element $x \in \mathbb{D}$ is then mapped to a distinct color $c(x) \in \mathbb{C}$, where $\mathbb{C}$ is also a discrete set of colors. For example, we can use a discrete colormap to map a software metric 'component failure' having values in $\mathbb{B} = \{true, false\}$ to the color set $\mathbb{C} = \{red, green\}$ respectively. Discrete colormaps can be classified as sequential, diverging or qualitative, as shown in Figure 6.14 which depicts several examples from the well-known ColorBrewer colormap construction application [14]. Clearly, software visualization applications can use both continuous and discrete colormap, depending on the type of software metrics at hand.

Several colormaps can be used within the same diagram visualization. For example, Figure 6.15 (bottom row) shows the same UML class diagram as in Figure 6.11, this time using two colormaps: the diverging colormap and the sequential colormap from Figure 6.14[2]. Using several colormaps has an advantage when different areas of interest in the same diagram have different types of metrics. In such a case, the identity, or meaning, of an area is also visible in the type of colormap used on it. To eliminate any interpretation ambiguities, this technique requires the usage of colormaps that do not share colors, like the diverging colormap (which uses shades of red, yellow, and green) and the sequential colormap (which uses shades of blue). In contrast, Figure 6.15 (top row) shows the same diagram and metrics, this time using a single colormap - the diverging one. Separating the different areas from each other is more difficult when using a single colormap than when using two different colormaps.

The second choice, or degree of freedom, is whether we interpolate colors at points situated between diagram elements, as described in Section 6.3.1, or not. To illustrate this choice, consider Figure 6.15 (right column) that shows the same diagram and metrics using the same colormaps as in Figure 6.15 (left column), this time without performing linear interpolation of colors. The color borders that appear between elements, corresponding to the approximate edges of the Voronoi diagram having elements as sites (see Section 6.3.1), are now clearly visible, whether we use one or several colormaps, as illustrated by the images. This has advantages and disadvantages, as follows. Only those colors that exist in the discrete colormaps are used, as there is no color interpolation. This is precisely what is required when using discrete colormaps. However, it may be argued

[2]The assignment of colormaps to areas is here arbitrary, done only for the purpose of illustrating the effect of using more than one colormap in the same image

that the color borders visually interfere with the texture patterns, thereby making it more difficult to follow the variation of a color, thus the variation of the depicted metric, over a given area. For continuous colormaps, such as the rainbow colormap, we can technically choose for interpolation, as the interpolated colors are part of the colormap. This will generate colors between diagram elements that do not correspond to actual data values, which is potentially confusing, but the resulting image will be arguably easier to follow visually. From the user evaluations done so far (see also Chapter 7, we have noticed no confusion or objections from the users, but this topic requires more investigation.

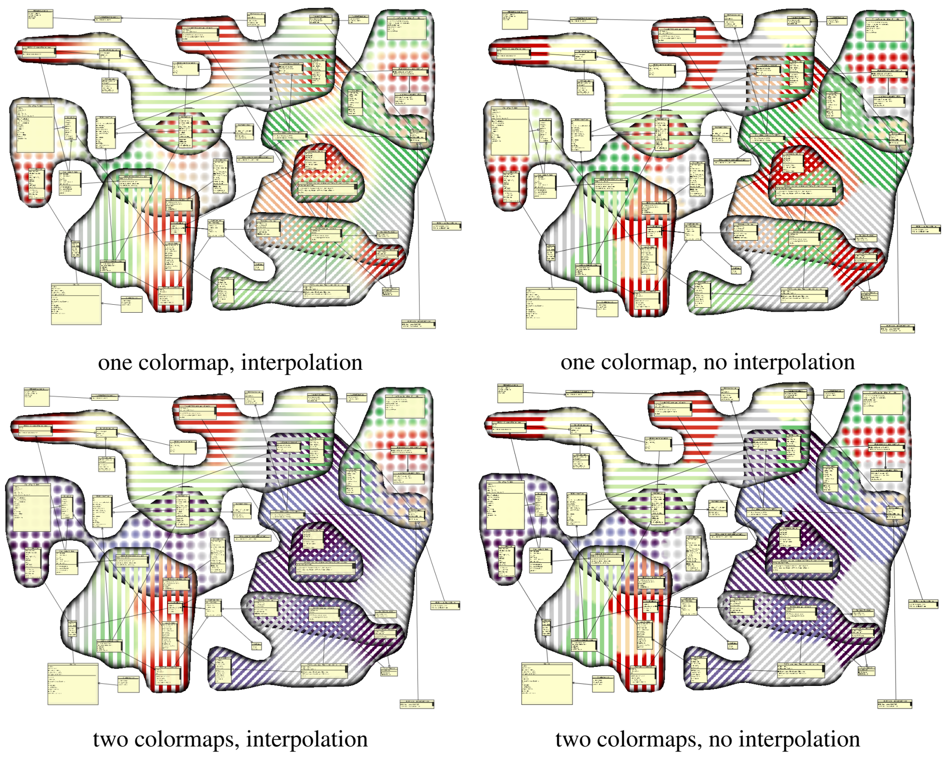

one colormap, interpolation
one colormap, no interpolation
two colormaps, interpolation
two colormaps, no interpolation

Figure 6.15: Effect of color interpolation and the usage of several colormaps when visualizing AOI-level metrics

The third and last degree of freedom is the choice of the colors in the colormap. Apart from application-driven considerations like which colors make more sense for which metric type, there is also an interaction of colors with the texture patterns and shading mechanisms described earlier in this chapter. To illustrate this, consider Figure 6.16 which renders a simple area of interest with one metric using a three-value sequential color scheme similar to the one shown in Figure 6.14. The top row shows the application of texturing and shading, whereas the bottom row also adds interpolation. Several observations can be made. First, we see that the texturing step considerably diminishes the actual number of pixels on which color is shown (in this case by roughly 50%). Since the colors in the colormap are quite similar visually, it becomes quite hard to tell the actual color that surrounds an element after texturing (top-middle image in the figure). If color interpolation is also done, then telling which color surrounds each element is even harder (bottom-middle image in the figure). Shading does not change this situation in worse or in better as it only affects pixels close to the area's boundary (right column in the figure). Concluding, we believe that, to obtain the best results with our texture-based method, one should choose colormaps that use strongly different hues, especially in the case when interpolation is applied. As a final observation on colormaps, several users remarked that using full-saturation hues is sometimes seen as distracting. To address this, we provide a global opacity control that allows users to set the overall opacity of all AOIs, thus smoothly navigate between 'bare' UML diagrams and diagrams with full-color 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.

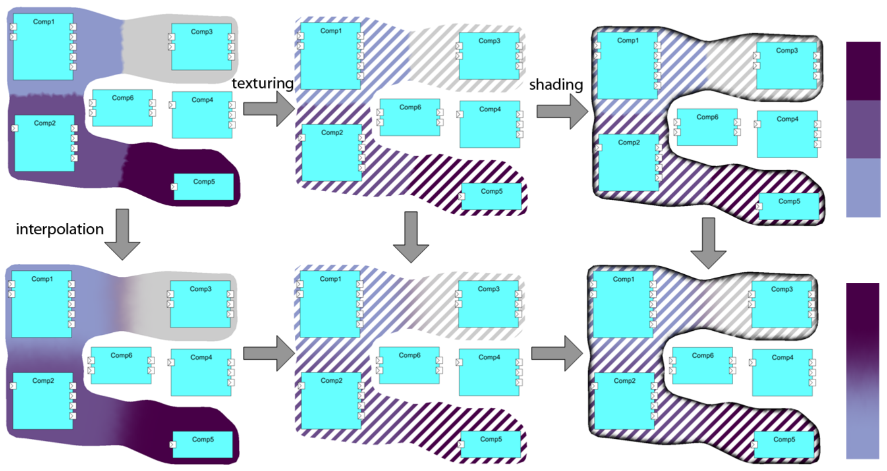


Figure 6.16: Applying sequential color scheme, interpolation, texturing and shading on a simple area example

As mentioned earlier in Section 6.3.3, shading has the added value of enhancing the perception of which elements are in which area and how areas overlap. Moreover, we believe that shading does not affect the perception of which metric values and element has. Figure 6.17 illustrates this. Here, the same UML class diagram and metrics as in Figure 6.15 is shown, without color interpolation, once with the rainbow colormap, and once with the diverging colormap discussed above, but without shading or drawing the contours of the areas. Seeing which metric values an element has in all areas it is contained within is arguably easy, and at least not more difficult than in Figure 6.15 which also showed the areas' borders. The rainbow colormap is better, as its hues are perceptually more different than the ones from the diverging colormap, as already discussed above. However, distinguishing the areas themselves is hard, since there are no borders drawn.

To summarize this discussion on colormaps and our metric visualization on AOIs:

- both discrete and continuous colormaps can be used, depending on the type of

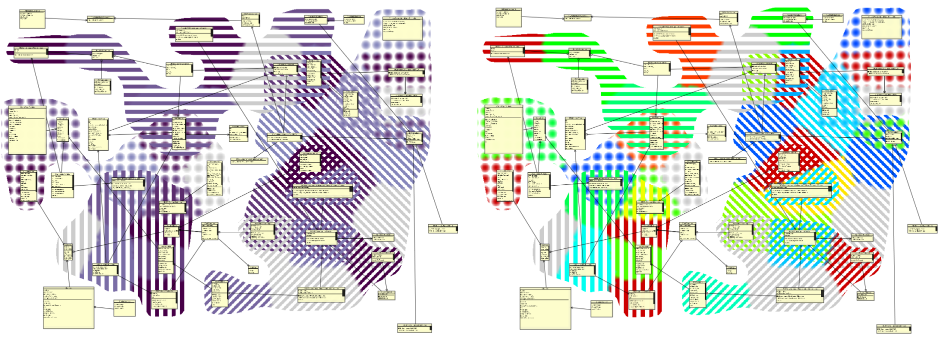


Figure 6.17: Metrics visualization without area shading using a sequential colormap (left) and the rainbow colormap (right)

metrics

- several colormaps can be used in one visualization as long as they have different colors
- discrete colormaps require switching off color interpolation
- interpolation may be confusing even for continuous colormaps, but it can easily be switched off
- colormaps using strongly different hues work best together with texturing
- shading enhances perception of areas and does not interfere with color mapping

6.6 Conclusion

In this chapter, we have presented a method to visualize metrics, defined on groups of elements represented as areas of interest, atop of software design diagrams. The key design decision in our proposed method is the rendering of metric information outside the extents of the diagram elements, and inside the extents of areas of interest, using a combination of textures, blending, and shading. The proposed method scales well in terms of the sizes of diagrams and areas of interest, and allows up to three overlapping areas with metrics at any given point. Moreover, the method combines well with the previously presented methods for drawing areas of interest (Chapter 3) and method-level metrics (Chapter 5), keeps the layout of a given UML-like diagram unchanged, and thereby concludes our investigation in the field of adding different types of metrics to software design diagrams.

Chapter 7

Applications in Industry

In this chapter we discuss the importance of verification of proposed theoretical ideas in industrial settings and benefits of validation and acceptance of new designed techniques. We present the main goals and constrains of a joint academic-industry project in which our proposed AOIs drawing technique was applied. We describe the case study which was considered in the project, our section of work in the project and collaboration with other project partners. Finally, we present the archived results.

7.1 Introduction

The central goal of the visualization techniques presented in this thesis is to help software engineers in understanding existing correlations between software architectures, areas of interest, and software metrics. In the previous chapters, we have presented several types of evaluations of the proposed visualization techniques. The largest body of work in this direction has been covered in Chapter 4, where we described a user study that produced both quantitative and qualitative assessments of the similarities of computer-drawn and hand-drawn areas of interest. In Chapters 5 and 6, we presented several smaller-scale case studies where our method-level and area-level metric visualizations, respectively, have been used to support several program comprehension tasks in reverse engineering.

Overall, these evaluations and case studies indicated that, in most cases, our visualizations are accepted, understood, and useful by software engineers. However, these studies do not fully answer the question of how our software visualizations would be accepted and integrated in the actual workflow of a typical software engineering project, along with other design and implementation techniques and practices. In other words, these studies do not fully close the gap between theory and practice. Doing this is quite challenging. Many additional problems exist in practice, like a great variability in actual requirements and working practices of industrial software engineers; time limitations for testing and verifying new tools and methods; the need to integrate a visualization tool with other tools; and the specificity of questions that need to be answered.

Because of all these, it is difficult to carry out experiments to validate a new technique, whether related to visualization or not, in an industrial setting. Furthermore, trying a new technique that emerges from a research setting in another context usually requires additional implementation or adaptation. Finally, visualization tools are not immediately recognized as valuable in a 'traditional' industrial software engineering context, where people are accustomed to work with command-line tools and only a small set of visual tools such as UML editors.

In this chapter, we describe our efforts to test a subset of our visualization techniques within such an industrial setting. In parallel with developing the AOI rendering method described in Chapter 3, we participated in the ITEA Trust4All project [41, 40], which involved partners both from academy and the software industry. In this context, we introduced our AOI rendering technique, implemented in an end-user tool for visualizing and editing UML diagrams and areas of interest. The tool was presented with the goal of helping the people involved in this project in a number of specific tasks related to their activities, as described further in this chapter. In the following, we describe our participation in the Trust4All project and overview the feedback obtained from the other (non-visualization-related) participants in the project as to the usefulness of our visualization techniques. Since the project took place during the phase of our research where only the AOI visualization was available, we present only feedback related to this technique. The method-level and area-level metric visualizations are evaluated separately as described in Chapters 5 and 6.

Section 7.2 describes the context of the project, the domain of the addressed problems, and the requirements we elicited for our work in the project. Section 7.5 presents a case study we have worked on, our UML-based AOI visualization tool, and the achieved results. Section 7.7 concludes the chapter with observations related to the acceptance of our visualization within this type of industrial context.

7.2 Context of the evaluation

The application of our proposed AOI drawing method was done in the context of the Trust4All project [41, 40]. The aim of this project was to improve the state-of-art in *component-based software development* (CBSD) by introducing new techniques for upgrading and extending embedded software systems, while the system is in use by a customer. CBSD techniques are briefly overviewed in Section 7.2.1. For this purpose, the component-based systems should be analyzed and results of the analysis should be represented in an effective way. Since these results contain both system architecture diagrams and groups of components related by their participation in different aspects, a natural application of our AOI visualization technique was possible.

To better understand the place of visualization in this project, we first briefly overview the basics of the component model used in the Trust4All project (Section 7.2.1) and the main goals of this project (Section 7.2.2). Based on these premises, we next describe the requirements for visualization of component-based systems which formed the basis of our work in this project (Section 7.3).

7.2.1 Component-based software development

One of the recent answers to the decrease in time-to-market and development costs of software products outlined in Section 1.1) is the use of component-based software development (CBSD). The basic idea of CBSD is that a system is assembled from pre-existing software components, thereby favoring the reuse of off-the-shelf software. In this context, a software *component* can be defined as: "a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." [97]. It is widely accepted that a software component should have the following properties:

- it should be designed and implemented for multiple use
- it does not expose its internal state to a system
- it provides well-specified interfaces, by which it can be bound to a system
- its dependencies on external resources are clearly specified
- its internal implementation should be encapsulated
- it communicates through its interfaces (provided and required)
- it can be substituted by some other component having compatible interfaces

In the Trust4All project, the ROBOCOP component model was used [38]. ROBOCOP stands for Robust Open Component Based Software Architecture. It was developed for use in middleware for consumer electronic devices, such as mobile phones, domotic devices, or industrial automation software. Its design is based on ideas from the previous CORBA [63] and Koala [114] component frameworks, which, in their turn, incorporate ideas from older component frameworks such as Microsoft's COM [61] and Sun's Enterprise JavaBeans [96].

Besides components, there are two other important concepts in the ROBOCOP framework: component models and component composition. A *component model* specifies the component development and deployment process, component implementation details, specification of component properties which include behavior, resource use, and context dependencies. It serves as a specification for composing individual components into an assembly. A ROBOCOP component is a set of possibly related models. Here, models can be represented in readable form, *e.g.* text, or as binary code. Examples of a component model are a security model, executable model or source code model. The models should be specified and packaged at the component development phase. A *component composition* is a set of instantiated components and bindings (connections) between their respective provided and required interfaces. Components are selected and composed by an architect in order to satisfy global system requirements, and according to the rules specified in the component model. For example, the executable model enforces rules specifying how actual functions in the source code are called when components interact which each other, while a security model specifies how a system's security depends upon the security of individual component interfaces in its composition.

The primary task that we supported in our visualization tool was the actual system creation by visual component composition. In this sense, our visualization tool implemented most of the functions typical to a UML editor, such as creation, deletion, connection, and disconnection of components on a 'component diagram'. However, we did not support the visualization of system deployment or of the communications with the underlying operating system or any hardware components. The component composition visualization is further detailed in Section 7.5.1.

7.2.2 The Trust4All project

The Trust4All project was the third step in a series of three ITEA projects in the area of CBSD. The first project, ROBOCOP, addressed the design and implementation of the ROBOCOP component model [38]. The second project, Space4U, added supplementary component models to the basic ROBOCOP framework, to specify resource-constrained (*e.g.* memory, processor, and power consumption) system design for embedded applications [39]. In each of these projects, six academic institutes and ten industrial companies from four European countries participated.

In Trust4All, the third and last project in this series, the goal was to add new mechanisms to the original ROBOCOP framework in order to specify and enforce security-related features to component-based systems whose software is dynamically extended and upgraded. Briefly put, this framework is a middleware layer built atop of the underlying operating system, which manages component creation and binding, registration, and deployment. Upon the dynamic downloading of a new component in a running system, the component framework ensures the proper working of the composed system by inspecting the various models of the downloaded component. Next, the framework attempts to optimize the overall system functioning by considering the requirements and provisions of all components (as specified in their models) and the resource constraints of the underlying hardware. Such computations are usually done in real-time.

To model these new dynamic aspects, a *trust model* was added to the ROBOCOP component model. The trust model captures software properties such as security, reliability and robustness. The main purpose of the trust model is to make these properties measurable. The component framework measures the properties according to rules predefined in the trust model and reacts depending on the computed results. For example, it can accept or deny a specific action on a new component, or emit other 'load-balancing' actions on existing components in the running system.

The second task that we supported in our visualization tool, along with the visual composition mentioned in Section 7.2.1, was to visualize the trust-related attributes of a component system. The main stakeholders of our visualization tool are the system architect, developers and testers, who need to analyze the changes in the system quality properties, *e.g.* trust, before and after any changes in the system structure due to new component deployments in a running system.

Visualization of trust is similar aspects to the visualization of other attributes, or metrics, of a software system, such as quality or complexity, which were discussed in the previous chapters. From the point of view of visualization, trust can be seen as a per-component, per-subsystem or per-entire-system numerical or categorical attribute or metric, which is computed by some methods outside of the scope of the visualization itself. In this sense, the tasks that trust visualization should address are:

- let users spot correlations and/or outliers of the trust values of specific elements and their relationship with the system structure;
- give an overview, as well as a detailed, understanding of the trust values of specific components, subsystems, or other system architectural elements.

However, for a visualization tool to be useful within this project, several requirements had to be fulfilled. In the following section, we detail these.

7.3 Visualization requirements

As mentioned above, a visualization tool in the Trust4All project should serve several tasks. In brief, this tool should support both the design and analysis of component-based systems along the lines mentioned in the previous sections. A large list of such functional requirements (FR) and non-functional requirements (NFR) were elicited from discussions with the project partners. In the following, we present only a small subset of these requirements, which are directly related to our work presented in this thesis. These are divided into two main classes:

- *R1*: Requirements related to obtaining insight into the quality attributes of a component diagram
- *R2:* Requirements related to the interaction (creation, editing, and navigation) with a component diagram

Table 7.1 lists the sub-requirements of class *R1*, as extracted from the actual requirement documents of the project. These are further categorized as functional (FR) and non-functional (NFR) requirements.

Table 7.1: Requirements on visualization of quality attributes

FR-11	The tool shall be able to visualize component compositions
FR-12	The tool shall be able to correlate and visualize components and their quality attributes
FR-13	The tool shall be able to store and retrieve the visualization results
NFR-11	Each visualization model entity should represent information unambiguously (by color, shape, texture, or text)
NFR-12	The tool should be configurable to work with different types of input information (for example, UML-like diagrams)
NFR-13	The set of supported layouts and views shall be extendable

Table 7.2 lists the sub-requirements of class *R2*, as extracted from the actual requirement documents of the project.

Table 7.2: Requirements on interaction with a component visualization

FR-21	The tool must be able to perform conditional selections of the components and their trust attributes
FR-22	The tool must be able to navigate through the levels and areas of the visualization model
FR-30	The tool should be integrated with data sources providing metrics computed on the analyzed system

In the following section, we overview the design of the entire tooling framework that was constructed to integrate the component-based system design with the data analysis required to compute the actual attributes which serve as inputs to our visualization, and finally the visualization itself. The actual use of our visualization method is covered further in Sections 7.5 and 7.5.1.

7.4 Overall framework design

To address the various tasks of system design, quality metrics computation, and metrics-and-structure visualization, several tools were constructed and integrated in a tooling framework. The overall framework architecture, outlined in Figure 7.1, shows the three main elements mentioned before: the system design tool, the system analysis and metrics computation tool, and finally the visualization tool.

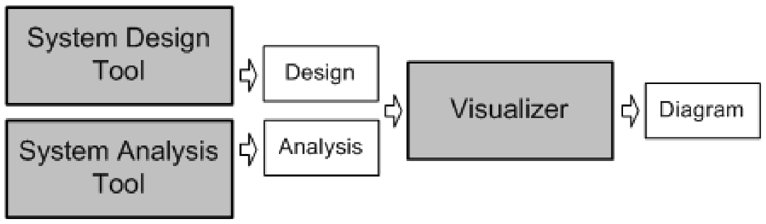


Figure 7.1: Architecture of the design, analysis, and visualization framework

Both the *design tool* and the *analysis tool* are provided by the CARAT toolkit, also developed as part of the Trust4All project [11, 12]. CARAT is a framework for design and performance analysis of real-time component-based software systems. It supports the complete design cycle and by means of three tools: a a component repository, a visual designer, and a simulator. The repository provides storage and retrieval of executables of software components and various models of software components and hardware blocks. The designer, implemented using the Eclipse framework [24], contains two editors for visual construction of component assemblies and hardware resource topologies with assigned deployment of the software components. Visual design is supported by means of drag-and-drop of components from the repository to a design canvas and point-and-click connection of provided and required interfaces, much like other visual application builders [105]. Figure 7.3 shows a typical component diagram for an actual system designed with this tool. The simulator uses the designed system model for static system analysis and predictions, which are stored as software metrics.

Our visualization tool comes at the end of a design and analysis cycle performed with the CARAT toolkit. This is illustrated in Figure 7.2), which shows a typical workflow in the design-simulation-visualization framework. The pre-design and design blocks represent respectively the construction of the component models and component assembly into a running system, both done with the CARAT toolkit. The execution of the resulting system is then simulated in the prediction block, which outputs various quality metrics, mapped to their corresponding components on the initial design diagram. These form the input for the last step, the visualization. In this step, the user can visually correlate the results of the system simulation with the initial design decisions, and, if desired, repeat the loop by adjusting its initial design decisions to optimize various quality metrics.

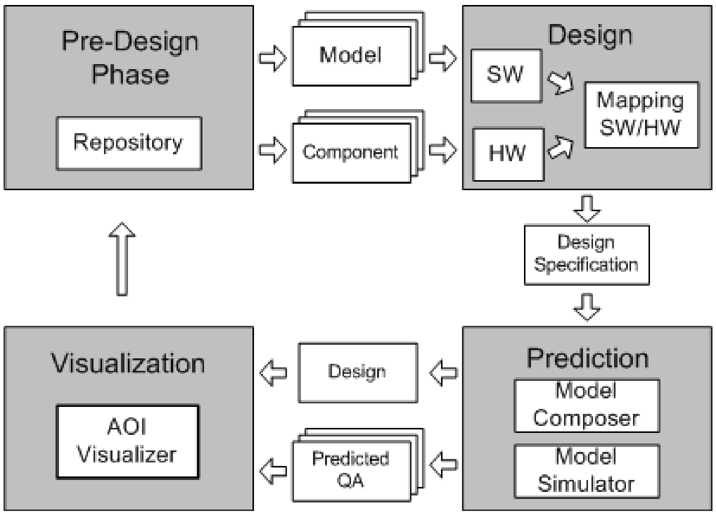


Figure 7.2: Software design/analysis life cycle covered by CARAT and AOI Visualizer

The actual operation of the integrated design, simulation, and visualization tools in this framework is illustrated next by means of a case study which was developed during the Trust4All project.

7.5 Case study: Car Media Center

Using the integrated framework presented in the previous system, a Car Media Center (CMC) software system was developed. This system contains several of the typical embedded software applications present in a modern car: GPS-based car navigation, radio and digital TV reception and display, and CD/DVD playback. Figure 7.3 is a snapshot from the CARAT designer tool showing the CMC system design consisting of 28 components and the connections between provided interfaces (pin-like icons at the left of the components) and required interfaces (icons at the right of the components).

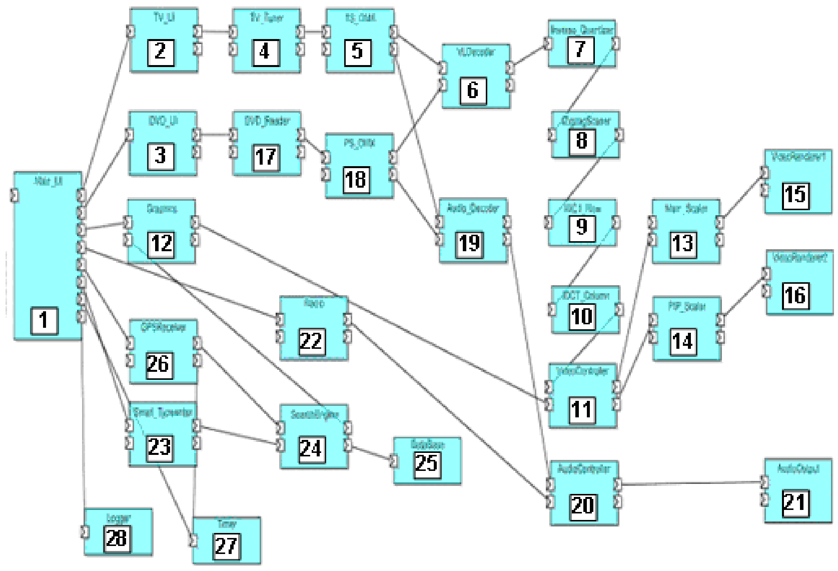


Figure 7.3: Car Media Center component-based design (CARAT *composer* snapshot)

The CMC has a dataflow-like design, with the following components. The Main UI component (1) receives user input by polling the buttons on the car dashboard. The TV UI (2) and DVD UI (3) components receive and process TV and DVD-related user commands. TV UI sends the currently selected TV channel to the TV Tuner (4). The transport bit stream of the chosen TV channel is sent to the TS DMX (5) component, which de-multiplexes the stream into video and audio. The video stream is next processed by several video filters: VLDecoder (6, variable length decoder), Inverse Quantizer (7), IZigzag Scanner (8, inverse zigzag scan), IDCT row and IDCT column (9, 10, inverse row/column discrete cosine transform). The decoded video stream is next sent to the VideoController (11) component, which specifies on which display to show the video. A second video stream comes to the Video-Controller from the Graphics (12) component carrying the graphical data (UI and navigation) coming from the Main UI component. The VideoController outputs two video streams to the Main Scaler (13) which scales images to display size or the PiP Scaler (14) which scales images to picture-in-picture format). Two VideoRenderer (15, 16) components perform the actual rendering. The audio path starts from the TS DMX (5) or DVDReader (17) and PS DMX (18) components, goes to the AudioDecoder (19) and AudioController (20), and ends up in the AudioOutput (21) component, which controls the car loudspeakers. AudioController also accepts the audio stream from the Radio (22) component and decides which of the two streams to play. The car navigation is implemented as follows. The user inputs an address via the Smart Typewriter (23) component. The address is next sent to the SearchEngine (24) component, which finds the desired location by querying the DataBase (25) component, compares it with the current car location received from the GPSReceiver (26), and computes the best driving path. The path and driving instructions are sent to the Graphics component for video rendering and to the AudioController component for voice messages. Finally, the Timer (27) and Logger (28) components perform system-wide synchronization and logging.

7.5.1 Visualization results

During the analysis phase of the CMC system described above, the CARAT simulation produced a number of software metrics were produced, such as CPU load, memory consumption, and component availability. Other metrics, such as the different types of vendors of the off-the-shelf software components involved in the design, were readily available from the component repository. A number of questions of the designers involved these metrics, as follows:

- How are component functions related to vendors?
- Which components are on the video or audio paths?
- Which components have user interface functions?
- Is performance-sensitivity related to functionality?
- How is availability related to functionality?

These questions relate well to our areas of interest. Indeed, we could define one separate area of interest for all components which fall into one of the categories induced by the above questions. Next, we could visualize the correlation of these areas of interest to answer the questions.

In total, the designers of the system defined seven areas of interest, as shown in Figure 7.4.

Area	Meaning of components in the area
A_1	Components produced by Vendor A
A_2	Components produced by Vendor B
A_3	Components on the video path
A_4	Components on the audio path
A_5	Availability-sensitive components
A_6	Performance-sensitive components
A_7	Interaction-sensitive (GUI) components

Figure 7.4: Areas of interest for the CMC architecture

For constructing the areas, we used thresholds on the metric values, based on values mentioned as relevant by the actual system designers. Introducing a threshold hides the actual metric values on the components, but, on the other hand, it simplifies understanding by producing a simple overview of the system.

Our users first tried to visualize these areas of interest using the standard metric icons provided by MetricView tool, which they were familiar with [107], by assigning different marker icon shapes and colors to every area. Components in one area thus share the same marker shape and color, which are chosen in some way so that they look different for different areas. The markers are also scaled to reflect the metric values before thresholding, but this aspect is not important in the following discussion. Figure 7.5 shows the result for three such areas, related to components shading the same vendor, components involved in the video and audio paths, and components which are sensitive to availability, performance, and user interaction aspects. As expected, this visualization is not very easy

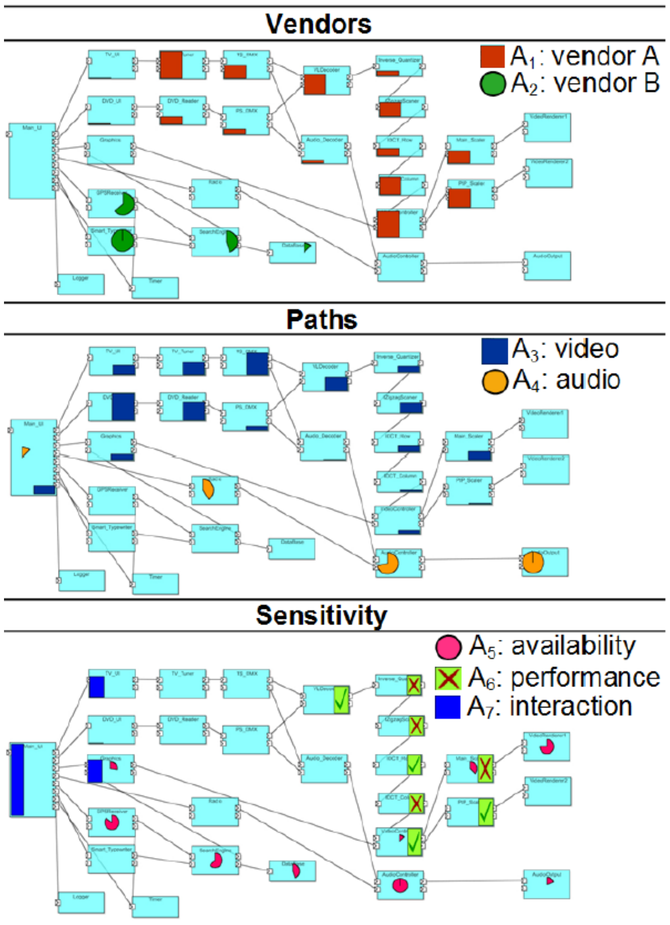


Figure 7.5: AOIs for the CMC system (shown with icons)

to follow, due to the difficulty in correlating small icons spread over a relatively large diagram.

Next, the users tried to visualize the same areas, this time using the inner-skeleton-based splatting method described in Chapter 3. Figure 7.6 shows the result, which uses the same area colors as for the markers in Figure 7.5. The areas are now easier to follow. Looking at the Vendors and Paths visualizations, for example, we see now easily that all video components (A3) come from vendor A (A1).

Still, this visualization has some problems. In the Paths view (Figure 7.6 middle), it is not quite clear whether the leftmost component is in both the video (A3) and audio (A4) areas. Also, in the Sensitivity view (Figure 7.6 bottom), it is not clear how the availability (A5) and performance (A6) areas overlap exactly. Moreover, the star-shaped form of the AOIs is somehow visually distracting. This suggested (at least to one user) there is something special about the element(s) located at the star center, which is of course not the case.

We used next our improved outer skeleton rendering method. The result is shown in Figure 7.7. In the middle image, we see a dark area around component 1 (leftmost)[1]. This says, as explained in Section 3.5.2, it is in two areas (i.e. video and audio). The advantage of this visualization is even clearer when we compare Figure 7.7 (bottom) with Figure 7.6(bottom). The dark areas show now easily the overlap of A5 (availability) with A6 (availability) and A7 (performance). Comparing the Sensitivity with the Vendors and Paths views answers further questions. We see that only video components (A1) are performance-sensitive (A6). The interaction-sensitive components (A7) are found only at the beginning of both video and audio paths (A3,A4). Only components from vendor B (A2) have availability-related problems (A5), except video component 11 which is from vendor A. Finally, we locate three interesting components (VideoController, MainScaler and PiP Scaler, i.e. 11,13, and 14 in Fig. 15) which are both performance and availability-sensitive.

Finally, let us mention that we can show diagrams, component metrics, and areas of interest together in a single view, if desired. Figure 7.8 illustrates this with a snapshot from the actual visualization tool used in this project. The various user interface controls support several navigation, selection, and searching functions, in line with the visualization requirements outlined earlier in Section 7.3.

Apart from the CMC system, we used our AOI visualization for several other component-based systems designed and simulated with the CARAT framework. A second example featuring a JPEG decoder was described earlier in Section 6.4.1. This second example also demonstrates the visualization of area-level metrics within the analysis of component-based systems.

7.6 Evaluation

Within the project, we conducted several sessions with designers to present our tool, demonstrate our visualization techniques, and get feedback. This development was originally driven by the specific project requirements listed in Tables 7.1 and 7.2. User feedback was important in several aspects, and allowed us to perform several improvements of our visualization method, as well as establish and refine the general requirements of the AOI rendering method (see Table 3.2), which we used after that throughout our work. These improvements, as well as other relevant user feedback elements, are discussed next.

[1]In the following, the component numbers refer to Figure 7.3

7.6.1 AOI construction

The initial technique used within our evaluations was the inner skeleton method (Section 3.4). From early stages, users noticed and reported several limitations of this technique, such as the inability to handle complex area configurations containing elements situated far away on the same diagram. Interestingly, the generation of ’false colors’ due to the blending of the actual area colors, which is discussed in Section 3.5.2, was not perceived as a problem by the users. This may be due to the fact that there were relatively few (under 5) areas on a diagram at a single time, and that the overlaps were, in the initial CMC case study, relatively small.

However, the inability of the inner skeleton areas to handle complex geometric configurations was clearly reported as problematic. This led us to the development of the outer skeleton method (Section 3.4). This method was immediately perceived as superior, which led us to its adoption and to the further refinements described in Section 3.6.

7.6.2 Interactivity and robustness

A requirement mentioned as essential by users of our visualization, even in its early stages, was interactivity. In detail, the possibility to perform both interactive re-layouts of the component diagram, *e.g.* by dragging the elements around with the mouse, and interactive changes in transparency and coloring, were mentioned as highly desirable by actual users. Together with this, robustness of the AOI rendering method, *i.e.* its ability to handle complex configurations without producing incorrect results, was mentioned as essential. This led to a number of further improvements to the AOI method (Section 3.6.3), as well as some of the computational optimizations mentioned throughout Chapter 3.

7.6.3 Area-level metrics

As the first results of our flat-shaded AOI allowed addressing some of the developers’ questions such as the ones mentioned at the beginning of Section 7.5.1, a set of refined questions appeared. For example, instead of ”is performance-sensitivity related to functionality?”, the question would be ”in what measure is performance-sensitivity related to functionality?”. These questions prompted our further research into ways to show the actual variation of a metric on an area of interest, and ultimately to the creation of the area-level metric visualization described in Chapter 6.

7.6.4 Smooth navigation

In the beginning, users of our AOI visualizations were not familiar with the concepts of ’areas of interest’. Moreover, we noticed that the visualization tool was also frequently used for other tasks than just answering questions related to areas of interest, such as browsing the system architecture. For this purpose, the possibility to smoothly navigate between a ’classical’ diagram-like view and a diagram view annotated by the colored areas of interest, by means of interactively tuning the areas’ transparencies, was one of the most used interaction features. This underlies our initial claim from Chapter 1 that users are more inclined to adopt a visualization tool when the tool allows them to incrementally add new features to an already known, familiar, visualization, such as the diagram view. In our case, this view was familiar from the CARAT designer tool (compare *e.g.* Figures 7.3 and 7.8).

7.6.5 Tool integration

Probably the most important requirement mentioned by the users of our visualization was its *integration* within the accepted workflow of the project, in this specific case the CARAT-based toolkit described in Section 7.4. This did not come as a surprise, as visualization tool integration is frequently mentioned as one of the main issues in tool adoption within the software industry [46, 83]. Here, as well as in many other contexts, this required a high amount of custom development and implementation work, more often than not unrelated to actual visualization issues, as well as a high amount of effort needed to understand the specific requirements of the users. However, without this effort, we are almost certain that it would not have been possible to convince the other project partners to consider visualization as a useful contribution to the overall project goals.

7.7 Conclusion

In this chapter, we have presented the experiences and lessons learnt when using the AOI visualization method introduced in Chapter 3 within the workflow of a large software engineering project containing a mix of research and industry participants. The overall conclusion, drawn after extensive discussions with the project participants during several working sessions spread over a period of two years, is that the AOI technique was perceived as useful and intuitive in understanding the correlation of a system's architecture with several (non)functional aspects, represented by areas of interest. This experiment also provided additional confirmation of our suppositions that the use of a known visualization metaphor, in our case the traditional node-and-link component diagram, and a high level of integration with other tools used in the established workflow, are essential elements that contribute to the acceptance and success of a visualization tool in an industrial software development context.

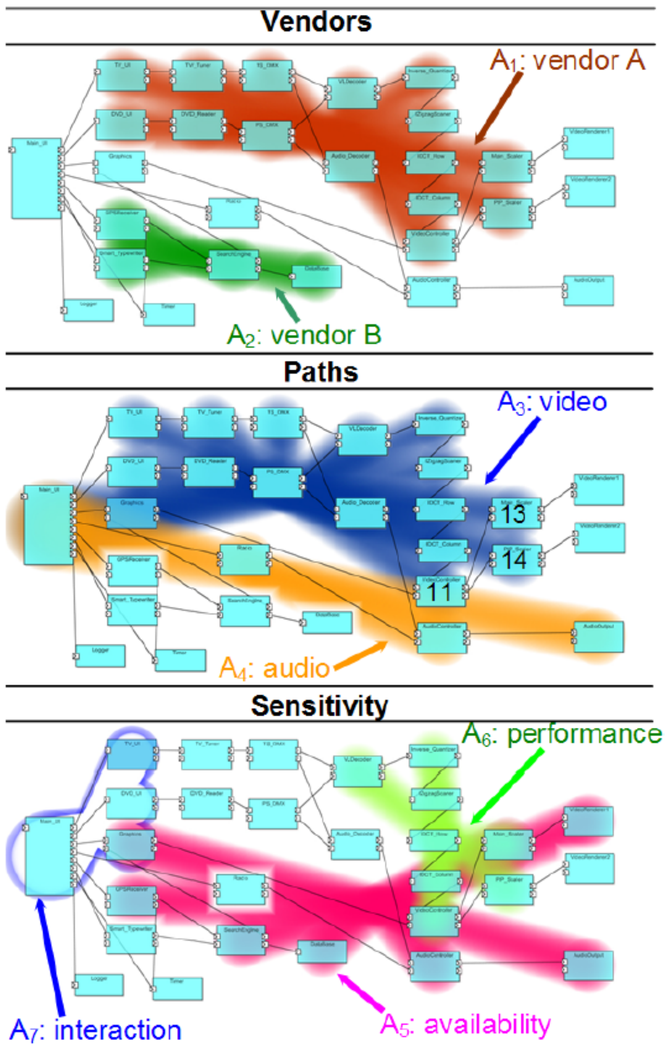


Figure 7.6: AOIs for the CMC system (shown with the inner skeleton rendering method)

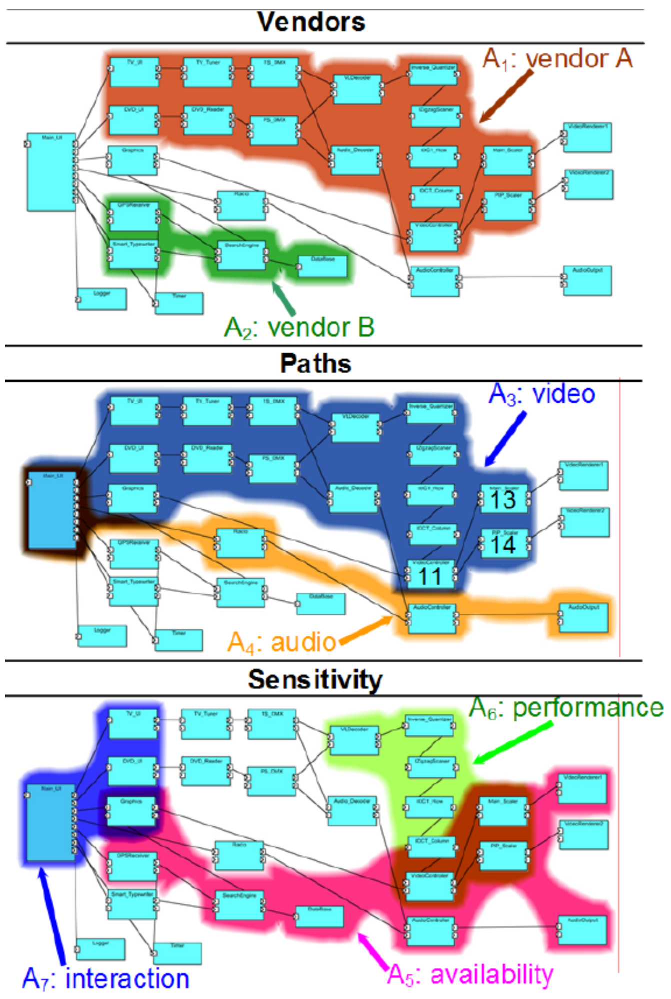


Figure 7.7: AOIs for the CMC system (shown with the outer skeleton rendering method)

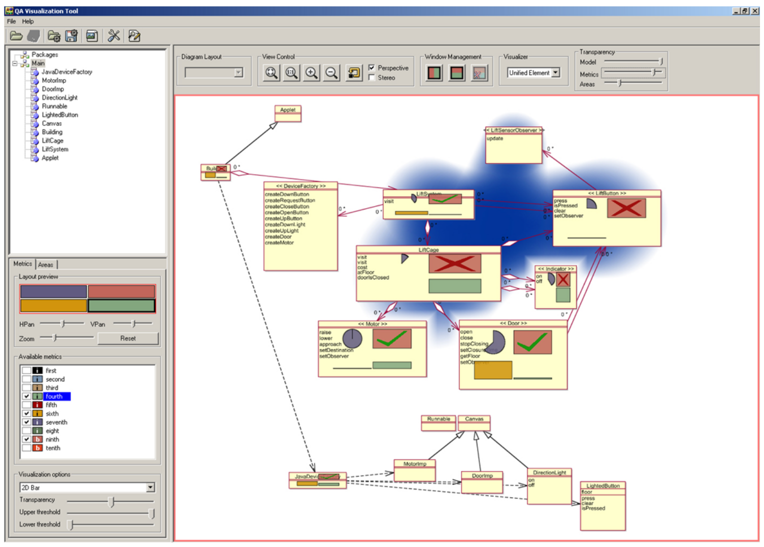


Figure 7.8: A snapshot of the area-and-metrics visualization tool

Chapter 8

Discussion

In this chapter, we provide a short discussion of the main results and findings obtained during the research which was described in detail throughout this thesis. As such, we do not repeat the specific conclusions drawn at the end of each of the previous chapters. The focus here is rather on outlining higher-level findings, or cross-cutting concerns, observed during the overall work in this thesis, which may have an influence on the continuation of this research.

8.1 Dimensions

In the beginning of this work, we identified three main requirements that an integrated method for visualizing metrics and software architectures should follow: UML-related look and feel, scalability, and understandability (Section 8.1). As such, the following discussion of our findings will also be structured along these requirements, or dimensions, with a focus on explaining the degree up to which these requirements have been fulfilled. Within each dimension, we shall consider the three main algorithmic contributions presented in this thesis: visualization of areas of interest, member-level metrics, and area-level metrics, as well as their possible interactions.

8.2 UML-like look and feel

The UML look-and-feel of all our visualizations appears to be well preserved by the techniques developed throughout our work. First and foremost, the basis of all our visualizations consists of a classical rendering of a UML diagram [107]. The only salient difference between this type of rendering and the ones produced by classical UML editors such as Rational Rose, Poseidon, Telelogic's Tau or MetricView, is the use of transparency and blending to (de)emphasize certain elements. However, in the context of diagram rendering, transparency is used sparingly. By default, there is a single transparency value for the entire diagram, which is close to one (opaque). In a single scenario, we use a separate low transparency value for the association edges of the diagram (Section 5.4.3). As an additional observation, we noticed that, when adding method-level metrics (Chapter 5), the transparency of the methods' textual names needs to be turned down so that the text does not interfere with the metric lens.

Another observation is the good fulfilling of the predefined layout constraint. That is, in virtually all use cases and experiments we did, we used in our visualizations the unchanged diagram layouts as we got them from earlier stages such as manual user layouts or input XMI files. The only case where we actually computed diagram layouts from scratch was in the reverse-architecting scenarios described in Section 5.4. However, we should stress that our work did not target the improvement of the quality of layouts. Some of these automatically computed diagram layouts were expectedly not optimal, and that we had to adjust a few either by hand or by changing the parameters of the layout engines used. In this context, it is important to note that these layout adjustments were *not* triggered by our visual additions to UML diagrams, such as metrics and areas of interest. Rather, the required adjustments were caused by problems in the *structural* comprehensibility of the produced diagrams. As such, we believe that our additional visualizations (metrics and areas) do not require layout changes for diagrams which already exhibit good structural comprehensibility.

Finally, we note that the rendering of method-level and area-level metrics, and of the areas themselves, leaves the look-and-feel of the UML diagrams largely unchanged. Information is added either inside element frames (method-level metrics) or between such frames (area-level metrics and areas). This, added to the fact that we can globally tune the transparency of all metrics and areas, makes our visualizations functionally look like *annotations* of UML diagrams. This is in strong contrast with the structure-and-metrics visualizations discussed in Chapter 2 (except MetricView that our work inherits from), which choose radically different layouts and shapes to show their information as compared to UML diagrams. As already stated, this is an important choice. We argue that UML-like visualizations favor acceptance and comprehensibility, at a possible expense in terms of scalability and freedom of constructing new, possibly more effective, visual representations. A large-scale longitudinal study for determining whether UML-like visualizations are indeed more effective than other types of metrics-and-structure visualizations would be needed. Such a study, however, is subject of future work.

8.3 Scalability

The main aspect of scalability we are interested in is the *visual* one. That is, we are interested in understanding what is the maximal size of diagrams and maximal number of areas of interest, metrics per element, and metrics per area, that we can visualize at one time. This aspect is strongly correlated with understandability (larger diagrams and more metrics arguably generate images which are harder to understand). We discuss understandability separately in the next section. Here, we focus strictly on algorithmic (computational) scalability of our proposed techniques.

From a computational perspective, the scalability of our improved rendering algorithm for *areas of interest* (Section 3.6) is sufficient for the targeted domains, *i.e.* typical UML diagrams having at most 100..200 elements and 10..20 areas of interest. Higher amounts of data are theoretically possible, but highly unlikely in practice on typical diagrams. Moreover, rendering more data at one makes little sense with our techniques, as understandability becomes a bottleneck (see Section 8.4).

The rendering complexity for a single area of interest is dominated by the cost of triangulating its contour (Section 3.5.2), in the case we render the area filled with a single color or when we use area-level metrics (Chapter 6). When sampling the area's contour with n points, this cost is $O(nlogn)$. The other costs, *e.g.* computing the nearest-element to a given contour point and the actual rendering cost, are much smaller (see Section 3.5.1). Overall, a careful implementation using spatial search structures, fast triangulation methods, and optimized OpenGL rendering code, delivers near-real-time performance for the typical datasets mentioned above. This is important as users can edit the diagram or change visualization options interactively to explore different analysis scenarios.

The rendering of *method-level metrics* (Chapter 5) is equally scalable. This process is linear in the number of methods visible on a class diagram times the number of metrics chosen for display for these methods. Since typical diagrams will show at most a few hundreds of such (method,metric) pairs, the rendering is fast. Moreover, the underlying table-lens technique used to visualize these pairs is proven to work in real time for data tables of tens of thousands of items [101].

Finally, the rendering of the *area-level metrics* (Chapter 6) is also scalable to our targeted datasets. For one area, the complexity of this method is again dominated by the triangulation cost, which is discussed above for the rendering of filled areas of interest. Apart from this, the other main computational task in the area-metrics technique is the Laplacian diffusion used to smooth out colors and compute the shading (). This cost is linear in the number of mesh points within the triangulated area. Overall, several area-level metrics can be rendered in subsecond time on complex diagrams.

8.4 Understandability

Understandability of a (software) visualization is naturally related to the tasks that the visualization is supposed to support. Strictly speaking, we cannot measure understandability *in general*, but should design either several understandability measurements for specific tasks, or factor out and evaluate a number of general understandability measures which are relevant for a set of tasks.

In this thesis, the most detailed analysis of understandability relates to the rendering of the areas of interest, which is done by means of a user study involving quantitative and qualitative aspects (Chapter 4). The considered tasks revolved around quickly detecting which elements are part of a given area and which areas contain a given element. Overall, the conclusions of this experiment are that our automatically-generated areas of interest are similar in visual quality and comprehensibility to good-quality areas drawn by human users, but they cannot match the quality of the best-drawn areas by humans. The main factors limiting the understandability of areas-of-interest relate to configurations when several areas visually overlap at a given place. Specifically, near-tangent contours cause problems in visually tracking the contour of an area of interest (Section 4.2.3). Such problems exist less in user-drawn configurations, as users tend to consider the already drawn areas when drawing a new one.

The algorithms presented in Chapter 3 can be extended to incorporate global optimizations that would reduce such contour crossings. For example, one could detect zones of contour overlaps using a distance function between contours similar to the one introduced in Section 4.3. Next, the contours placement could be optimized by using a cost function that incorporates all quality aspects, such as high-angle contour intersections, contour smoothness, and the correct inclusion of elements. However, this potential solution would require additional study, as it is not clear design a cost function which is effective in encoding the desired quality metrics and also very efficient to optimize.

For the *method-level metrics*, understandability is related mainly to the tasks of finding elements having (large or small) outlier metric values for several methods; and finding metrics which are correlated, that is, show similar values for the same methods of different diagram elements. Here, we noticed that the encoding of the metric values in both metric bar length and hue (Section 5.4.1) is considerably more effective than using just one of the encodings. This is not surprising, as the visual space allocated to one single metric can be as small as a few tens of pixels when zooming out on large diagrams. In terms of number of methods shown, understandability is good up to several tens of methods per class, due to the table lens metaphor. In terms of number of metrics, understandability becomes limited once we try to visualize more than three metrics in the same time on the same diagram. The reason seems to be due to a layout constraint: in many (class) diagrams, element icons are relatively thin but tall, *i.e.* they allow displaying many methods, but not long signatures. Tables having many columns will thus have little space on the horizontal axis for showing each column (see *e.g.* Figure 5.10). If we relax our original constraint and allow changing the diagram layout by making the elements wider, visualizing more than three metrics with good results should be possible.

For *area-level metrics*, understandability is related mainly to the tasks of finding elements having (large or small) outlier metric values in several areas; elements having different, or similar, values, within all areas they are part of; and getting an idea over the evolution of a single metric over an entire area. For these tasks, the understandability of the visualizations produced by our method is limited by two main factors. First, there is the choice of the colormap and interpolation method. As discussed in Section 6.5.2, using colormaps with strongly different hues is recommended when color interpolation is preferred (the suitability of the latter being a matter of debate). Second, area metrics become hard to understand at places where more than three areas overlap. This seems to be inherent to our method of combining colors at one point using texture weaving (Section 6.3.2). It may be possible to design different visualizations that address the above-mentioned tasks for area-metrics with a higher degree of understandability. However, this is subject for future work.

8.5 Conclusions

From the perspective of our three main requirements, we can conclude that the techniques presented in this thesis allow us to construct combined visualizations of software system diagrams and software metrics defined on groups of elements and element members, and that these visualizations maintain a UML-related look and feel, scale well to the typical datasets available in software architecting, and are understandable from the perspective of typical program comprehension tasks. Moreover, our techniques require a minimal user intervention, *i.e.*, work automatically, do not pose additional requirements or limitations on the data in the treated diagrams, and can be added in a non-intrusive way to classical UML-like software visualizations.

Chapter 9

Conclusion

9.1 Summary

In this thesis, we have studied the creation of visualizations of combined software architecture diagrams and software metrics. For the architectural data, we have used a UML-like representation of the software structure, as UML diagrams are arguably well understood and widely accepted by the main target group of our visualizations, the software architects and designers involved in creating and understanding complex software systems. Besides the structural data which is inherent to UML diagrams, such as entities and relationships between entities, we also considered a new data type: groups of elements that are related to one concern, or aspect, in a given system design, called *areas of interest*. We have presented methods to render areas of interest on software diagrams in a way that imitates the style actual human users would draw such annotations on paper or whiteboard diagrams. For metrics, we have presented ways to visualize metrics defined on diagram element members, such as class methods, as well as metrics defined on elements involved in areas of interest.

Overall, the presented techniques work as *annotations* on existing UML-like diagrams, rather than proposing entirely new ways for combining metrics with diagrams, such as, for example, 3D visualizations. Using transparency and blending, all visualized items, *i.e.* metrics and areas, can be emphasized, visually pushed in the background, or completely removed from a given UML diagram visualization, without changing the visualized diagram's layout. This enables users to smoothly navigate between a classical, well-understood, diagram view, and a diagram view annotated with additional metric or area-of-interest information.

We have evaluated the proposed visualizations on several levels. First, we have conducted a user study that compared the understandability of our automatically-generated areas of interest with user-drawn areas of interest, and elicited a number of algorithmic improvements that brought our visualizations close to good human drawings. Secondly, we have used our visualizations in several case studies focusing on understanding various aspects of software architectures, with a focus on reverse engineering and maintainability. Finally, we have used the prototype visualization tool that was constructed during this work in the framework of an academic-industry collaboration, and observed the reactions of actual users with respect to the proposed visualization methods.

9.2 Directions of Future Work

There are several possible directions of future work from the results presented here. In the following, we outline these directions, ordered on their perceived potential to produce useful results for the overall task of helping users understand the correlation of structures with metrics on diagrams.

9.2.1 Metrics on relations

All techniques presented in this thesis consider the visualization of metrics defined on *entities* in the entity-relationship model that underlies a software architecture diagram. However, many such metrics exist also for relationships [51]. For example, for associations that indicate function calls, we can consider the number of times a function is called; the length (duration) of the call; whether the function is virtual, overloaded, static, or a remote procedure call. For associations that indicate data members, we can consider the number of times the member is accessed, and whether the access is a read or a write. For inheritance relations, we can consider the type of inheritance (public, private, or protected); or the amount of interfaces which is defined, specialized, or used within the inheriting class.

It is challenging to consider how to render several such metrics atop of relations in a UML-like diagram, especially if we want to keep our constraints of not modifying a given layout. The problem is not trivial, since typical diagrams can have a large number of relations, whose visual line-like representations can intersect several times. A different way would be to explore visualizations where relations become first-class citizens [71], and see how such methods can be combined with classical UML diagram visualizations.

9.2.2 Relations and areas

Our areas of interest are, so far, strictly defined in terms of entities. However, it may be desirable to explicitly include relations within areas of interest (Section 3.8.2). This would be useful, for example, in the case when certain elements are involved within an area of interest from the perspective of a given relation, and involved in other areas, or no area, from the perspective of another relation. Since a UML-like diagram has a single representation for a given element, we may need ways to make the connection between areas and relations visually explicit on the diagram.

Different routes are possible to address the above goal. First, one could constrain the geometric shapes and positions of areas of interest so that they make explicit which relations between their contained elements these areas include. However, using solely this technique, the visual connection between an area and the relations it includes may be not sufficient. Secondly, one could design additional visual cues to mark the inclusion of relations within a given area (or areas), such as using specific shading and texturing between that area's contour and the included relations. This technique may give good results, however it can also potentially cause undesired occlusions and visual clutter. Finally, one may consider a way to draw the areas of interest which is radically different fro our current Venn-Euler diagram-like shapes, with the aim of making the inclusion of both entities and relations within an area visually explicit.

9.2.3 Visual scalability

As outlined in Chapter 8, there are several limitations to the visual scalability of our visualization methods. Improved results and increased usability can be obtained if we augment the number of method-level and area-level metrics displayed at a given time on a diagram; and if we reduce the visual clutter created by multiple overlapping areas of interest. For the latter goal, it seems possible to adapt or reuse many of the geometric and shape processing methods in existence, such as feature-preserving smoothing or medial axes, so that we construct geometric shapes for the areas which are closer to those quality criteria which are perceived as important by users. Assuming that we have reliably detected which are these quality criteria, and that we can quantify them, the main concern here is to maintain the current robustness and speed of our visualization methods, which are indispensible for their actual use in practice.

9.2.4 Different application domains

Essentially, there is little that makes our work here strictly applicable to software engineering diagrams. As outlined in Chapter 2, metrics and areas of interest occur also in datasets emerging from other domains, such as social network and organization analysis. It would be interesting to see whether the techniques presented in this thesis are applicable to these other domains and datasets, and which modifications may be needed. In particular, it would be challenging to consider relational diagrams defined over 'concrete' spaces, such as geographical maps, and to extend the idea of areas of interest to incorporate the inclusion of entire zones from such maps, rather than the more restricted definition we have now which focuses only on inclusion of nodes from a graph.

As a less technical, but still suggestive, example in this direction of larger applicability of our areas of interest to different domains, we choose to end this thesis with an image taken from a movie [86] which shows a doctor sketching a diagram showing relations between various diseases, an image which is strikingly similar to our own areas of interest on software architecture diagrams.

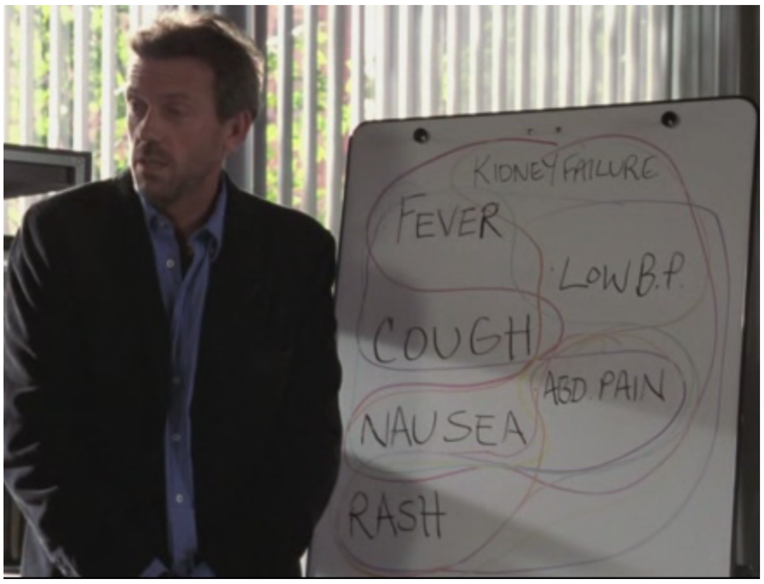


Figure 9.1: Areas of interest present in different application domains

Bibliography

[1] A. KUHN, P. LORETAN, O. N. Consistent layout for thematic software maps. In *WCRE 08* (2008), IEEE Press, pp. 209–218.

[2] AOSD. Aspect-oriented software development. `http://www.aosd.net/`.

[3] ARANDA, J., ERNST, N., HORKOFF, J., AND EASTERBROOK, S. A framework for empirical evaluation of model comprehensibility. In *Proc. Intl. Workshop on Modeling in Software Engineering (MiSE)* (2007), pp. 7–15.

[4] ARYA, S., MOUNT, D., NETANYAHU, N., SILVERMAN, R., AND WU, Y. An optimal algorithm for approximate nearest neighbor searching. *J. of the ACM 45* (1998), 891–923.

[5] AT & T. GraphViz. `www.graphviz.org`.

[6] BALZER, M., AND DEUSSEN, O. Level-of-detail visualization of clustered graph layouts. In *APVIS '07* (2007), IEEE Press, pp. 133–140.

[7] BERRETTI, S., BIMBO, A. D., AND PALA, P. Retrieval by shape similarity with perceptual distance and effective indexing. *IEEE Transactions of Multimedia 2*, 4 (2000), 225–239.

[8] BISCHOFBERGER, W. Sniff+: A pragmatic approach to a c++ programming environment. In *Proc. USENIX C++ Conference* (1992), pp. 67–81.

[9] BLOOMENTHAL, J., AND BAJAJ, C. *Introduction to Implicit Surfaces*. Morgan Kaufmann, 1997.

[10] BOBROWSKA, A. A framework for empirical evaluation of model comprehensibility. In *Proc. SOFSEM* (2005), Springer, pp. 72–81.

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

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

[13] BORLAND INC. Together modeling tool. `www.borland.com/together`.

[14] BREWER, C. A. Color use guidelines for mapping and visualization. In *Visualization in Modern Cartography* (1994), Elsevier Science, pp. 123–147.

[15] CHEN, X., AND PLIMMER, B. CodeAnnotator: Digital ink annotation within eclipse. In *Proc. OZCHI* (2007), ACM, pp. 211–214.

[16] CORBI, T. A. Program understanding: Challenge for the 1990s. *IBM Systems Journal 28*, 2 (1999), 294–306.

[17] COSTA, L., AND CESAR, R. *Shape Analysis and Classification: Theory and Practice*. CRC Press, 2001.

[18] COSTA, L. F., AND CESAR, R. M. *Shape Analysis and Classification: Theory and Practice*. CRC Press, 2001.

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

[20] DE BERG, M., CHEONG, O., KREVELD, M. V., AND OVERMARS, M. *Computational Geometry: Algorithms and Applications (3rd ed.)*. Springer, 2008.

[21] DI BATTISTA, G., EADES, P., TAMASSIA, R., AND TOLLIS, I. G. *Graph Drawing*. Prentice Hall, 1999.

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

[23] DUCASSE, S., AND LANZA, M. The class blueprint: Visually supporting the understanding of classes. In *IEEE Trans. on Software engineering, vol.31, nr.1* (2005), IEEE Computer Society.

[24] ECLIPSE. The eclipse foundation. `http://www.eclipse.org/`.

[25] EICHELBERGER, H. Aesthetics of class diagrams. In *Proc IEEE VISSOFT* (2002), IEEE Press, pp. 23–31.

[26] EICHELBERGER, H. *Aesthetics and Automatic Layout of UML Class Diagrams*. Univ. of Würzburg, Germany, 2005. PhD thesis.

[27] FEKETE, J. D. The infovis toolkit. In *Proc. InfoVis* (2004), IEEE, pp. 167–174.

[28] FENTON, N., AND PFLEEGER, S. *Software Metrics: A Rigorous and Pracical Approach*. Chapman & Hall, 1998.

[29] FERENC, R., MAGYAR, F., BESZEDES, A., KISS, A., AND TARKIAINEN, M. Columbus reverse engineering tool and schema for C++. In *Proc. Intl. Conf. Software Maintenance (ICSM)* (2002), IEEE, pp. 172–181.

[30] GALLAGHER, K., HATCH, A., AND MUNRO, M. Software architecture visualization: An evaluation framework and its application. *IEEE Trans. Softw. Eng. 34*, 2 (2008), 260–270.

[31] GAVRILA, D. M. Multi-feature hierarchical template matching using distance transforms. In *Proc. ICPR* (1998), pp. 439–444.

[32] GDT. GDTOOLKIT: A graph drawing toolkit. `www.dia.uniroma3.it/~gdt/`.

[33] GILL, N., AND GROVER, P. Component-based measurement: A few useful guidelines. *ACM SIGSOFT Software Engineering Notes 28* (2003).

[34] GOLDSTINE, H., AND VON NEUMANN, J. *Planning and coding of problems for an electronic computing instrument*. Part II, volume I of a report prepared for the U.S. Army Ord. Dept., 1947.

[35] GTAUMAN, K., AND DARRELL, T. Fast contour matching using approximate earth mover's distance. In *Proc. CVPR* (2004), pp. 220–227.

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

[37] IBM. *Rational Rose Modeling Tool*. `www.306.ibm.com/software/rational`.

[38] ITEA. The *Robocop* project. `http://www.hitech-projects.com/euprojects/robocop/`.

[39] ITEA. The *Space4U* project. `http://www.win.tue.nl/san/projects/space4u/`.

[40] ITEA. The *Trust4All* project. `www.win.tue.nl/trust4all`.

[41] ITEA. International technology education association. `http://www.iteaconnect.org/`.

[42] J. J. VAN WIJK, A. T. Enridged contour maps. In *Proc. IEEE Visualization* (2001), IEEE Press, pp. 69–74.

[43] JACKSON, M. *Principles of Program Design*. Academic Press, 1975.

[44] JANSSEN, A., AND BOERBOOM, F. *Fact extraction, querying and visualization of large C++ code bases (Master's thesis)*. Eindhoven University of Technology, the Netherlands, 2006.

[45] KLEMOLA, T., AND RILLING, J. Modelling comprehension processes in software development. In *Proc. Intl. Conf. On Cognitive Informatics (ICCI)* (2002), IEEE Press, pp. 329–337.

[46] KOSCHKE, R. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance: Research and Practice 15*, 2 (2003), 87–109.

[47] KROGSTIE, J. Evaluating uml using a generic quality framework. In *UML and the unified process* (2003), Idea Group Inc., pp. 1–22.

[48] LANZA, M. Codecrawler - lessons learned in building a software visualization tool. In *Proc. of 7th European Conference on Software Maintenance and Reengineering* (2003), IEEE Computer Society, p. 409.

[49] LANZA, M., AND DUCASSE, S. Understanding software evolution using a combination of software visualization and software metrics. In *Proc. of LMO* (2002).

[50] LANZA, M., AND DUCASSE, S. Polymetric viewsa lightweight visual approach to reverse engineering. In *IEEE Trans. on Software engineering, vol.29, nr.9* (2003), IEEE Computer Society.

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

[52] LEMKE, I., AND G.SANDER. Vcg: Visualization of compiler graphs. In *Tech. Report 12/94* (1994), Universität des Saarlandes, Saarbrücken, Germany.

[53] LIKERT, R. A. A technique for the measurement of attitudes. *Archives of Psychology 140* (1932).

[54] LINDLAND, O. I., SINDRE, G., AND SOLVBERG, A. Understanding quality in conceptual modeling. *IEEE Software* (1994).

[55] LITTLEFAIR, T. C and C++ code counter. `sourceforge.net/projects/cccc`.

[56] LOMMERSE, G., NOSSIN, F., VOINEA, L., AND TELEA, A. The *Visual Code Navigator*: An interactive toolset for source code investigation. In *Proc. InfoVis* (2005), IEEE, pp. 24–31.

[57] LORENSEN, W. E., AND CLINE, H. E. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics, (Proc. ACM SIGGRAPH) 21*, 4 (1987), 163–169.

[58] MARCUS, A., FENG, L., AND MALETIC, J. 3D representations for software visualization. In *Proc. ACM SoftVis* (2003), p. 2736.

[59] MARIN, M., VAN DEURSEN, A., AND MOONEN, L. Identifying crosscutting concerns using fan-in analysis. *ACM Trans. on Software Engineering and Methodology 17* (2007).

[60] MARINESCU, C., MARINESCU, R., MIHANCEA, P. F., RATIU, D., AND WETTEL, R. iPlasma: An integrated platform for quality assessment of object-oriented design. In *Proc. of the 21st IEEE International Conference on Software Maintenance (ICSM)* (2005), pp. 265–272.

[61] MICROSOFT. The component object model. `http://msdn.microsoft.com/en-us/library/ms694363(VS.85).aspx`.

[62] MIHANCEA, P. F., GANEA, G., VEREBI, I., MARINESCU, C., AND MARINESCU, R. McC and Mc#: Unified C++ and C# design facts extractors tools. In *Proc. of the 9th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)* (2007), IEEE Press, pp. 103–111.

[63] MOWBRAY, T., AND ZAHAVI, R. *Essential Corba*. John Wiley and Sons, New York, 1995.

[64] NASSI, I., AND SHNEIDERMAN, B. *Flowchart techniques for structured programming*. SIGPLAN Notices, 1973.

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

[66] PAGE-JONES, M. *What every programmer should know about object-oriented design*. Dorset House Publishing, 1995.

[67] PARR, T., AND QUONG, R. Antlr: A predicated-ll(k) parser generator. *Software - Practice and Experience 25*, 7 (1995), 789–810.

[68] PLIMMER, B., AND GRUNDY, J. Beautifying sketching-based design tool content: issues and experiences. In *Proc. AUIC* (2005), Australian Comp. Soc., pp. 31–38.

[69] PLIMMER, B., GRUNDY, J., HOSKING, J., AND PRIEST, R. Inking in the IDE: Experiences with pen-based design and annotation. In *Proc. VL/HCC* (2006), IEEE, pp. 111–115.

[70] PREFUSE. The prefuse information visualization system. `http://prefuse.org`.

[71] PRETORIUS, A., AND VAN WIJK, J. Visual inspection of multivariate graphs. *Computer Graphics Forum (Proc. Eurographics/IEEE EuroVis) 27*, 3 (2008), 967–974.

[72] PRIEST, R., AND PLIMMER, B. RCA: Experiences with an IDE annotation tool. In *Proc. CHINZ* (2006), ACM, pp. 53–60.

[73] PURCHASE, H. C., CARRINGTON, D. A., AND ALLDER, J.-A. Empirical evaluation of aesthetics-based graph layout. *Empirical Software Engineering 7*, 3 (2002), 233–255.

[74] PURCHASE, H. C., CARRINGTON, D. A., AND ALLDER, J.-A. Graph layout aesthetics in uml diagrams: User preferences. *Journal of Graph Algorithms and Applications 6*, 3 (2002), 255–279.

[75] PURCHASE, H. C., COLPOYS, L., AND CARRINGTON, D. A. Uml collaboration diagram syntax: an empirical study of comprehension. In *Proc. VISSOFT* (2002), pp. 13–22.

[76] PURCHASE, H. C., WELLAND, R., MCGILL, M., AND COLPOYS, L. Comprehension of diagram syntax: an empirical study of entity relationship diagram notations. *International Journal of Human-Computer Studies 61*, 2 (2004), 187–203.

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

[78] RENIERS, D., AND TELEA, A. Tolerance-based feature transforms. In *Advances in Computer Graphics and Computer Vision* (2008), vol. 4, Springer, pp. 187–200.

[79] RILLING, J., AND MUDUR, S. P. 3d visualization techniques to support slicing-based program comprehension. *Computers and Graphics 29*, 3 (2005), 311–329.

[80] RUMBAUGH, J. Notation notes: Principles for choosing notation. *Journal of Object-Oriented Programming 12*, 4 (1999).

[81] SCHROEDER, W., MARTIN, K., AND LORENSEN, B. *The Visualization Toolkit - An Object-Oriented Approach to 3D Graphics (3rd edition).* Kitware, Inc., 2003.

[82] SCIENTIFIC TOOLWORKS INC. *Understand* for c++. `http://www.scitools.com`.

[83] SENSALIRE, M., OGAO, P., AND TELEA, A. Classifying desirable features of software visualization tools for corrective maintenance. In *Proc. ACM SOFTVIS* (2008), pp. 87–90.

[84] SETHIAN, J. *Level set methods and fast marching methods*. Cambridge Univ. Press, 1999.

[85] SHEWCHUK, J. R. Triangle: Engineering a 2d quality mesh generator and delaunay triangulator. In *Proc. Applied Computational Geometry* (1996), ACM Press, pp. 124–133.

[86] SHORE, D. House M.D. In *Movie series 2004-2008* (2008).

[87] SIDDIQI, K., AND PIZER, S. *Medial Representations: Mathematics, Algorithms and Applications*. Springer, 2008.

[88] SMITH, J. M., AND STOTTS, D. SPQR: flexible automated design pattern extraction from source code. In *Proc. Automated Software Engineering (ASE)* (2003), IEEE, pp. 215–224.

[89] SPENCE, R. *Information Visualization.* ACM. Press, 2006.

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

[91] SPRENGER, T. C., BRUNELLA, R., AND GROSS, M. H. H-BLOB: A hierarchical visual clustering method using implicit surfaces. In *Proc. IEEE Visualization* (2000), IEEE, pp. 61–68.

[92] STANDISH, T. A. An essay on software reuse. *IEEE Trans. Software. Eng. 10*, 5 (1984), 494–497.

[93] STOREY, M.-A. Shrimp views: An interactive environment for exploring multiple hierarchical views of a java program. In *Proc. of the 5th international conference on Aspect-oriented software development* (2006), ACM Press, pp. 146 – 157.

[94] STROUSTRUP, B. *The C++ Programming Language (3rd ed.).* Addison-Wesley, 2004.

[95] STRZODKA, R., AND TELEA, A. *Generalized Distance Transforms and skeletons in graphics hardware*. Springer, 2004.

[96] SUN. Enterprise javabeans. `http://java.sun.com/products/ejb/`.

[97] SZYPERSKI, C. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, 1997.

[98] T. PANAS, R. BERRIGAN, J. G. A 3d metaphor for software production visualization. In *Proc. of the Seventh International Conference on Information Visualization* (2003), ACM Press, p. 314.

[99] TAUBIN, G. Geometric signal processing on polygonal meshes. In *EUROGRAPHICS STAR Reports* (2000).

[100] TELEA, A. An open architecture for visual reverse engineering. In *Managing Corporate Information Systems Evolution and Maintenance (ed. K. Khan)* (2004), Idea Group Inc., pp. 343–364.

[101] TELEA, A. Combining enhanced table lens and treemap techniques for the visualization of large data tables. In *Proc. EuroVis* (2006), IEEE Press, pp. 13–20.

[102] TELEA, A. *Data Visualization: Principles and Practice*. A. K. Peters, 2008.

[103] TELEA, A., MACCARI, A., AND RIVA, C. An open toolkit for prototyping reverse engineering visualizations. In *Proc. VisSym* (2002), ACM Press, pp. 241–250.

[104] TELEA, A., AND VOINEA, L. SOLIDFX: An interactive reverse-engineering environment for C++. In *Proc. CSMR* (2008), pp. 320–322.

[105] TELEA, A., AND WIJK, J. J. V. VISSION: an object oriented dataflow system for simulation and visualization. In *Proc. IEEE VisSym* (1999), Springer, pp. 95–104.

[106] TELELOGIC. *Telelogic Tau Modeling Tool.* www.telelogic.com/products/tau.

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

[108] THERON, R., SANTAMARIA, R., GARCIA, J., AND GOMEZ, D. Overlapper: visualization of interconnected social groups. In *Proc. Graph Drawing (tool competition)* (2008), Springer.

[109] THOMAS, J. J., AND COOK, K. A. *Illuminating the Path: The R&D Agenda for Visual Analytics*. National Visualization and Analytics Center, 2005.

[110] TILLEY, S., WONG, K., STOREY, M., AND MÜLLER, H. Programmable reverse engineering. *Intl. J. Software Engineering and Knowledge Engineering 4*, 4 (1994), 501–520.

[111] TOM SAWYER, INC. The tom sawyer graph layout software suite. http://www.tomsawyer.com.

[112] VAN HAM, F., VAN DE WETERING, H., AND VAN WIJK, J. Interactive visualization of state transition systems. *IEEE Transactions on Visualization and Computer Graphics 8*, 4 (2002), 319–329.

[113] VAN LIERE, R., AND DE LEEUW, W. Graphsplatting: Visualizing graphs as continuous fields. *IEEE Transactions on Visualization and Computer Graphics 9*, 2 (2003), 206–212.

[114] VAN OMMERING, R., VAN DER LINDEN, F., KRAMER, J., AND MAGEE, J. The koala component model for consumer electronics software. In *IEEE Trans. Software Engineering* (2000), vol. 33, IEEE Press, pp. 78–85.

[115] VAN WIJK, J. J., AND VAN DE WETERING, H. Cushion treemaps: Visualization of hierarchical information. In *Proc. InfoVis* (1999), IEEE, pp. 73–78.

[116] VOINEA, L., AND TELEA, A. A framework for interactive visualization of component-based software. In *Proc. EUROMICRO* (2004), IEEE Press, pp. 567–574.

[117] VOINEA, L., AND TELEA, A. Multiscale and multivariate visualizations of software evolution. In *Proc. SoftVis* (2006), ACM, pp. 115–124.

[118] WARE, C. *Information Visualization, Second Edition: Perception for Design*. Elsevier, 2004.

[119] WETTEL, R., AND LANZA, M. Visual exploration of large-scale system evolution. In *WCRE 08* (2008), IEEE Press, pp. 219–228.

[120] WIGG, D. CPP_Parser: A C++ grammar for ANTLR. `http://www.antlr.org/grammar/list`.

[121] WONG, P. C., AND THOMAS, J. J. Visual analytics. *IEEE Computer Graphics and Applications 24*, 5 (2004), 20–21.

[122] WOO, M., NEIDER, J., DAVIS, T., AND SHREINER, D. *OpenGL Programming Guide, 3rd edition*. Addison-Wesley, 2001.

[123] WUST, J. *SDMetrics*: The software design metrics tool for *UML*. `www.sdmetrics.com`.

[124] YEUNG, L., PLIMMER, B., LOBB, B., AND ELLIFFE, D. Levels of formality in diagram presentation. In *Proc. OZCHI* (2007), ACM, pp. 311–317.

List of Publications

The work presented in this thesis has been partly covered by the following publications, listed in chronological order:

H. BYELAS, A. TELEA Visualization of Areas of Interest in Component-Based Architectures, *Proceedings of the EUROMICRO Conference on Software Engineering and Advanced Applications*, Dubrovnik, Croatia, IEEE Press, 2006, pp. 160-169
Related chapters: 3, 7

E. BONDAREV, M. CHAUDRON, H. BYELAS, P.H.N. DE WITH A Toolkit for Design and Performance Analysis of Real-Time Component-Based Software Systems, *Proceedings of the International Conference on Software Engineering Advances*, Papeete, Tahiti, IEEE Press, 2006, pp. 4-8
Related chapters: 7

H. BYELAS, A. TELEA Visualization of Areas of Interest in Software Architecture Diagrams, *Proceedings of the ACM symposium on Software visualization*, Brighton, United Kingdom, ACM Press, 2006 (best paper award and cover image on Proc. ACM SoftVis'06), pp. 105-114
Related chapters: 3, 7

H. BYELAS, A. TELEA Visualization of Quality Attributes on Software Architectures, *Proceedings of Scientific ICT-Research Event Netherlands*, Delft, Netherlands, 2007, (poster), pp. 127
Related chapters: 3, 7

H. BYELAS, A. TELEA Evaluating Visual Realism in Drawing Areas of Interest on UML Diagrams, *Proceedings of 14th Annual Conference of the Advanced School for Computing and Imaging*, Heijen, The Netherlands, 2008, pp. 265-272
Related chapters: 4

H. BYELAS, A. TELEA Towards Visual Realism in Drawing Areas of Interest on Software Architecture Diagrams, *Journal of Visual Languages and Computing*, vol. 10, no. 28, Elsevier, 2008, pp. 110-128
Related chapters: 4

H. BYELAS, A. TELEA The Metric Lens: Visualizing Metrics and Structure on Software Diagrams, *Proceedings of Scientific ICT-Research Event Netherlands 08*, Amsterdam, Netherlands, 2008, (poster)
Related chapters: 5

H. BYELAS, A. TELEA Texture-based visualization of metrics on software architectures, *Proceedings of the 4th ACM symposium on Software visualization*, Ammersee, Germany, ACM Press, 2008, (poster), pp. 205-206
Related chapters: 6

A. TELEA, H. BYELAS Visual Software Analytics for Assessing the Maintainability of Object-Oriented Software Systems, *Journal of Interaction, Intelligence and Information*, vol. 8, no. 1, 2008, pp. 43-62
Related chapters: 5, 6

H. BYELAS, A. TELEA The Metric Lens: Visualizing Metrics and Structure on Software Diagrams, *Proceedings of the 16th Working Conference on Reverse Engineering*, Antwerp, Belgium, IEEE Press, 2008, (tool demo paper), pp. 339-340
Related chapters: 5

A. TELEA, H. BYELAS, L. VOINEA Architecting an Open System for Querying Large C and C++ Code Bases, *South African Computer Journal*, vol. 41, no. 1, 2008, pp. 43-56
Related chapters: 3, 5, 6

A. TELEA, H. BYELAS, L. VOINEA A Framework for Reverse Engineering Large C++ Code Bases, *Electronic Notes in Theoretical Computer Science (special issue on Proc. Software Quality and Maintainability (SQM) 2008)*, 2009, Elsevier, pp. 143-159
Related chapters: 3, 5, 6

H. BYELAS, A. TELEA Visualizing Multivariate Attributes on Software Architectures, *Proceedings of the 13th European Conference on Software Maintenance and Reengineering* (doctoral symposium), Kaiserslautern, Germany, IEEE Press, 2009, pp. 335-338
Related chapters: 3, 4, 5, 6

H. BYELAS, A. TELEA Visualizing Metrics on Areas of Interest in Software Architecture Diagrams, *Proceedings of the Pacific Visualization Symposium*, Beijing, China, IEEE Press, 2009, (to appear, April 2009)
Related chapters: 6

H. BYELAS, A. TELEA, Texture-Based Metrics Visualization on Software Architecture Diagrams, *Proceedings of 15th Annual Conference of the Advanced School for Computing and Imaging*, The Netherlands, 2009 (to appear, May 2009)
Related chapters: 6