Visual Software Analytics for Assessing the Maintainability of Object-Oriented Software Systems

Heorhiy Byelas and Alexandru Telea

Institute of Mathematics and Computer Science, University of Groningen,
Nijenborgh 9, 9747 AG Groningen, the Netherlands
h.v.byelas@rug.nl, a.c.telea@rug.nl

Abstract

*Mésurer la maintenabilité des systèmes logiciels est une composante essentielle du développement logiciel moderne. Cette activité est traditionelemment executée en extrayant des metriques du code source par des techniques 'fouille de données'. Pour des systèmes peu connus, mesurer la maintenabilité doit être étroitement combiné à la compréhension du logiciel. Nous proposons pour cette tâche l'*analytique visuelle logicielle*, une nouvelle combinaison de la visualisation intéractive et fouille de données adaptée au code source. Nous présentons une application pour la compréhension et estimation de la maintenabilité du logiciel orienté objet, avec deux contributions. En premier lieu, la* table lens *superpose des métriques calculées sur des methodes en dessus d'une diagramme de classe UML, en aidant les corrélations métrique-métrique et métrique-structure. En deuxime lieu, la légende métrique permet de construire des analyses impliquant des différentes métriques, gammes de valeurs, et projections visuelles. Une implémentation a été conçue pour produire plusieurs analyses partant du code C++ et qui produisent des visualisations combinées métrique-diagramme.*

Mots-clés *: Analytique visuelle, visualisation du logiciel, maintenance du logiciel*

Abstract

Assessing the maintainability of software systems is an essential component of modern software development. Traditionally, this activity is performed by extracting source code metrics using data mining techniques. When the analyzed system is little known, maintainability assessment must go hand in hand with software understanding. We advocate for this task software visual analytics*, a new combination of interactive visualization and data mining focused at software code. We present an instance of software visual analytics for the understanding and maintainability assessment of object-oriented software. Our contributions are twofold. First, the metric lens visualizes method-level*

code metrics atop of traditional UML class diagrams, which supports metric-metric and metric-structure correlations. Second, the metric legend allows 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. We present an implementation in a full-fledged system and describe several analyses starting with real-world C++ code and ending with combined diagram-metrics visualizations.

***Key-words**: Visual analytics, software visualization, software maintenance*

1 INTRODUCTION

Software is everywhere. It is continuously developed by an estimated 15 million engineers worldwide [4], in a hierarchy of activities, ranging from requirement gathering, specification, and design, to implementation, debugging, testing, and maintenance. Software maintenance had become an essential component of the software life-cycle, accounting for over 80% of the entire efforts of the software life-cycle. Studies over 20 years, from [31] to [8], show that understanding software code accounts for over half the development effort. Software continuously evolves, as described by the laws of software evolution [2, 14], which only increases its complexity, and thus the understanding and maintenance effort. Overall, understanding software is hard, as it is large, complex, and abstract [16].

Since understanding is such an important part of software maintenance, a large effort has been invested in methods and tools that assist this task. From a usage point of view, such methods range from nearly fully automated data/mining approaches that extract maintainability and code quality metrics from software repositories [15] up to sophisticated interactive visualization systems [17].

A difficult, but important, task is understanding a system by persons who have not been involved in its development. Such situations occur *e.g.* when companies insource third-party software and have to assess its maintainability, or when new members join a team working on a large legacy project. In such cases, the primary (and often only) information is the source code itself.

In this paper, we focus on a specific class of software: object-oriented (OO) code bases written in the C++ language. OO software is highly structured, a favorable attribute in development, maintenance, and understanding in general. However, C++ software is well-known as being also highly complex, posing an increased burden on understanding. Although several methods and tools exist for understanding such software, there are still limitations in providing a high integration of data mining and data presentation (visualization) capabilities, which limits their applicability.

A well-accepted way of reasoning about an OO system is to represent it on a higher (design or architectural) level of abstraction. In this context, UML

diagrams are a conventional, well-accepted choice to represent the system on a design level. While diagrams show the software elements and their relationships, *i.e.* the system structure, software attributes, encoded into various software metrics, convey important insights in a system's properties, *e.g.* quality, maintainability, and modularity. In the following, we shall focus on *code-level metrics*, computed directly on the source code at method level.

Combining code metrics and diagram information in a single representation is an effective way to help several activities, 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 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 combination of code metrics and structural (diagram) information should comply with several requirements:

1. show the system structure and code metrics in a *single* representation, to simplify metric-structure correlations;

2. show *several* metrics in the same time, which may have different value ranges, to simplify metric-metric correlations;

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

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

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 paper, we propose such a solution, consisting of two new correlated techniques: the *metric lens* and the *metric legend*. First, we review related work on software architecture and software metrics visualization (Sec. 2). Section 3.1 introduces the data model we use in our structure-and-metric visualization and explains how the data is mined from source code. Section 4 presents our new structure-and-metric visualization: the *metric lens*, which adapts and extends the well-known table lens technique for software datasets (Sec. 4.1); and the *metric legend*, which lets users both specify analyses interactively and interpret the created visualizations (Sec. 4.2). Section 5 outlines the implementation of our techniques. Section 6 presents three case studies of our techniques in understanding a real-world software system. Section 7 presents and discusses our results. Finally, Section 8 concludes the paper.

2 RELATED WORK

Related work in the area of comprehending the structure and quality metrics of complex object-oriented code bases falls roughly in two classes: presentation of the *structure* of oject-oriented code, and presentation of the code *metrics*.

2.1 Generic Software Visualizations

Several visualization tools exist that provide varying degrees of customization and level-of-detail in presenting structural information extracted from source code. We call these techniques *generic*, as they are not constrained to a particular visual layout of the system structure. Static code structure is typically visualized as annotated graphs by tools such as Rigi [32], CodeCrawler [17], Mondrian [21], or SoftVision [33]. Metrics are most often presented separately using tabular views [42]. Having recognized the need to display metrics and structure in the *same* view, to facilitate metric-structure correlations, several researchers have taken the way of mapping the metric values to diagram element (icon) visual attributes, such as shape, color, size, or texture. For example, Wettel and Lanza use a cityscape metaphor: structure is shown using 2D treemaps, while metrics are shown using the third dimension (height) and color [39]. Similar techniques are presented by other authors [20, 7], ranging from 3D desktop graphics up to powerful, sophisticated virtual reality solutions [25]. An extensive overview of software visualization techniques is provided by Diehl [10].

2.2 UML-Based Visualizations

A different class of techniques uses a two-dimensional UML (or UML-like) diagram layout to visualize structure, instead of the generic graph layout or treemap techniques mentioned above. Arguably, UML diagrams are less compact and offer less layout freedom than generic graph layouts, so are less scalable. However, UML layouts have several advantages. UML diagrams are widely accepted and well understood in the software industry [29, 5]. Secondly, 2D renderings remove several problems of 3D visualizations, such as occlusion and orientation difficulties [38]. Last but not least, software engineers often desire to use a prescribed layout for system structure, *e.g.* a carefully hand-designed UML diagram, rather than an automatically generated layout [6] Recently, several attempts have been presented for extending basic UML diagrams with additional attributes. Byelas and Telea add metrics atop UML diagrams using colored icons and textures [35, 6]. Lanza *et al.* visualize metrics by mapping these to the sizes and colors of classes in UML-like diagrams [17].

However, the above methods can be improved in several directions. Mapping software metrics to visual attributes of graph nodes (size, color, texture,

lighting etc) works best for class or component-level metrics, but does not scale to the finer-grained level of *method* metrics. Adding metric icons to UML-based visualizations [35] has the advantage of using an accepted structural visualization (UML), but cannot show method-level metrics. Finally, 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.

3 A SOFTWARE ANALYTICS PIPELINE

As outlined in Sec. 1, our aim is to have an *integrated* 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 to specify their questions in an easy, interactive way.

Our proposed solution consists of a pipeline of operations (Fig. 1), 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 as 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 [36, 41]. 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 paper.

Our software analytics pipeline is detailed next.

3.1 Data extraction

Our visualization target, in the terminology of [26], 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 real-valued software metrics $\mu_1, \ldots, \mu_n$, each having a given value-range $\mu_i \in [m_i, M_i] \subset \mathbb{R}$. The UML diagrams and software metrics are extracted directly from source code, as described next.

Since our input is just the C++ source code of the system under study, we must extract our UML model from this code. For this, we use a so-called *tolerant* C++ parser able to understand incorrect and incomplete code, *e.g.* code with missing headers, to compute a partial abstract syntax tree (AST) of the input code. By partial, we mean that the computed AST may lack

some code constructs which cannot be parsed correctly due to the code's incompletness or incorrectness. Out of the several C++ parsers in existence which deliver such information, we could succesfully use an ANTLR-based parser [27, 23] using the C++ grammar described in [40], or SOLIDFX, a standalone heavyweight C++ analyzer [34]. The ANTLR-based C++ parser 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.

After the basic classes and class-relationships are extracted from the source code, the user specifies a division of the code base into subsystems by assigning source files to each subsystem. Next, one UML diagram for each subsystem is created, containing the classes in that subsystem's files. This step has to be manual, as the subsystem information is inherently context-dependent, so it cannot be always mined automatically from source code.

The second data extraction step computes the software metrics, both at class and method level. Class metrics include the typical number-of-methods, number-of-bases, inheritance-depth, and number-of-overriden-methods [18]. Method metrics include the lines-of-code (LOC), lines-of-comment-code (COM), and McCabes cyclomatic complexity (MVG) metrics, among others. In the following, we focus mainly on the method-level metrics, since it is for these that we provide our novel visualizations (Sec. 4.1 and 4.2). An important design decision was to separate the metrics computation from the architectural data extraction. For the architecture extraction, we use the ANTLR-based parser or SOLIDFX, as outlined above in Sec. 3.1. For metrics, we use the CCCC lightweight extractor [22] or the Understand software analysis tool [30]. Separating the two 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 [3]: we use a combination of existing tools, whenever available, to extract and combine the complementary information needed for our final analysis. The extracted metrics are saved in a XML-based database, where each row describes a class or method, and each column a particular metric. Depending on the completeness of the extraction, the values of several metrics can miss for specific methods.

The extracted UML model and metrics is fed to the visualization step of our software analytics pipeline. This step, and the two novel techniques it introduces, are described next.

4 VISUALIZATION DESIGN

Our combined structure-and-metrics visualization, the second part of our software analytics pipeline (Fig. 1, uses two views: the *metric lens* and the *metric legend*. The views are tightly coupled (Fig. 2). The metric lens com-

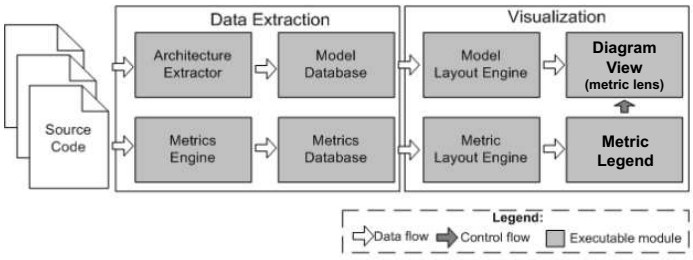

Figure 1: Architecture of the metrics-and-structure visualization pipeline

bines a classical UML viewer with a visualization of method-level metrics using an enhanced version of the well-known table-lens technique [28]. 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 metrics displayed in this view.

As mentioned in Sec. 1, the goals of our two-view solution are:

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

We next describe how the metric lens (Sec. 4.1) and the metric legend (Sec. 4.2) meet these goals.

4.1 Metric Lens

The basis of the metric lens technique is a traditional UML class diagram, which displays all textual methods[1] within each class frame (Fig. 3). Atop of this image, we display metrics using a *table model*, where the rows are methods met_i and columns are metrics M_j [2]). Each table cell shows one metric value using different icons. Missing metric values (Sec. 3.1) have no icon. All metric tables of all displayed classes can be sorted on various criteria, *e.g.* the method names or metric values, enabling different analyses, as discussed next in Sec. 6). The metric icon table can be placed within the class frames (Fig. 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. 3 c), which does not occlude the method names but yields a less compact layout.

Each metric value in a table cell is shown using a metric *icon*. We first used here the same design as in [35], *i.e.* a number of general-purpose icons

[1] Data members, or data fields, are treated identically

[2] We strongly recommend viewing this paper in color

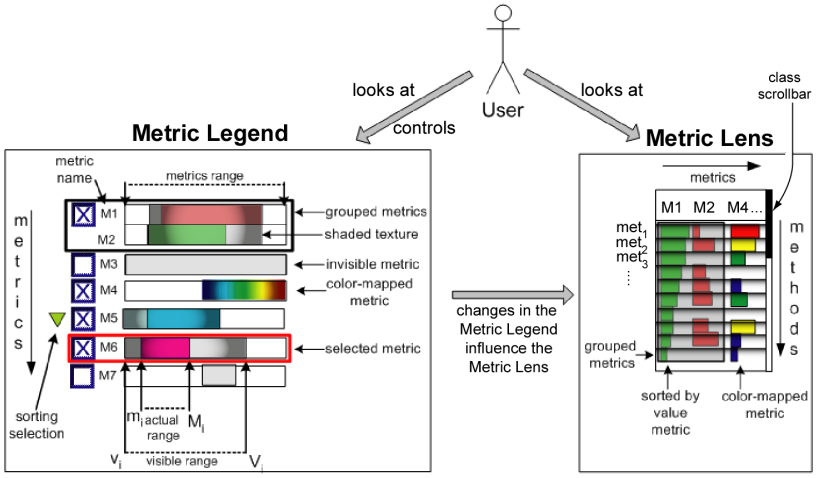

Figure 2: Structure-and-metrics visualization design: metric legend (left) and metric lend (right)

such as 2D and 3D bars, pie charts, and cylinders, scaled by metric values. However, this design does not scale well, since a UML diagram can easily have hundreds of methods, several tens per class, and each method can have several metrics.

To address this problem, we use a modified version of the table-lens technique [28]. 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 (see Sec. 4.2 next). Secondly, we provide two independent zoom mechanisms:

- *diagram-level:* this zooms the entire diagram, *i.e.* the UML layout, metrics, and method names;
- *class-level:* this zooms 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. Class-level zooming, in combination with class contents sorting, allows navigating between seeing the entire contents of each class, as a set of colored bar graphs (when zoomed out), and seeing the individual method signatures (when zoomed in). When the class frame size cannot accomodate all methods and metric icons, we display a scrollbar at the right of each class to allow scrolling through its contents (Fig. 2 right). We also implemented two enhancements as compared to the original table-lens [28]: First, we modulate the methods' text opacity by the class zoom level, so that class-zooming effectively interpolates between

a classical 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 (Fig. 2 right). This creates a subtle contrast that visually separates each method when the class contents are zoomed out and the text is not visible.

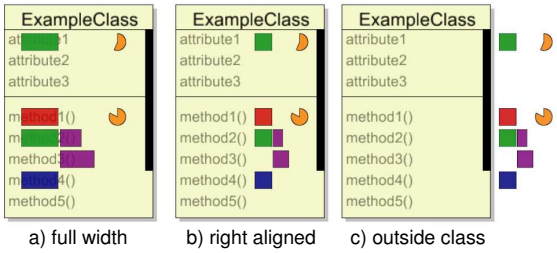

Figure 3: Metric layout options

4.2 Metric Legend

When visualizing several metrics over large diagrams, inferring the exact metric values from the metric bar sizes and/or colors can be very hard. Moreover, how to visually compare metrics which are defined over totally different value ranges; and how to specify which metrics to compare in a given scenario? We address these goals using a new visualization: the *metric legend* widget. This widget has two roles. First, it lets users interactively control which metrics from the data model are mapped to the metric lens, and how (Sec. 4.1). Second, it acts like a legend for interpreting the metric lens.

4.2.1 Basic design

The metric legend has a compact tabular structure (see Fig. 2 left for a schematic view and Fig. 4 from an actual snapshot). Each metric μ_i of the data model is a row in the widget. For each metric, a checkbox shows whether it is *visible*, *i.e.* shown with colored bars in the metric lens; the metric's *name*, *e.g.* $M1 \ldots M7$ in Figs 2 left and 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 Fig. 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 that of $M6$.

The bar colors indicate how each metric is shown in the table lens: Gray denotes metrics 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 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 Fig. 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.* $M4$. Clicking on the legend widget allows changing the color mapping from flat to rainbow.

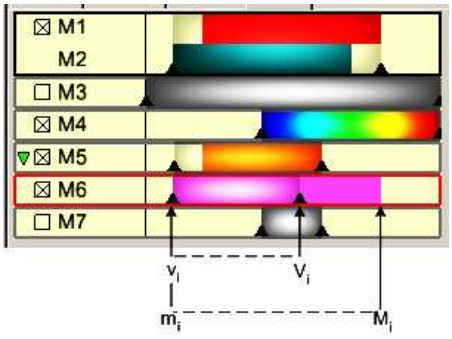

Figure 4: Metric legend (actual tool snapshot)

4.2.2 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 Fig. 4) over the range bar of the desired metric. 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 Sec. 6.

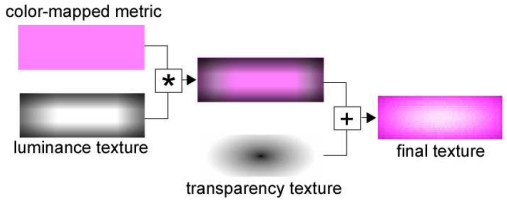

Figure 5: Texture design for the metric legend

Besides the range sliders, 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. 4 we see that the visible range $[v_i, V_i]$for metric $M6$ is approximately the lower half of its actual range $[m_i, M_i]$. We construct this shaded texture as follows (see also Fig. 5). First, we build a parabolic luminance texture dark at the borders and bright in the center (similar to the so-called shaded cushions used by [24]), 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 [24], 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. 4 are an example of the latter.

4.2.3 Grouping metrics

In practice, different software metrics may have completely unrelated meanings and ranges. For example, it does 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* in the metric legend. 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. 2 left and right, respectively.

All in all, the metric legend compactly specifies and explains the metric lens values in the diagram view. First, we can select which metrics to visualize from a potentially large precomputed set, by enabling their checkboxes. Second, we can compare actual metric ranges and metric values by comparing their bar lengths and colors, respectively. Tooltips in both the metric lens and legend indicate the actual metric values under the mouse. Third, we can see what value the color of a given icon in the metric lens actually *means*, by looking it up in the metric legend. Finally, we can specify which metrics make sense to compare visually by grouping them.

5 Implementation Details

We implemented the software analytics pipeline described so far in a fully integrated reverse-engineering tool aimed at C++ code bases. As outlined in Sec. 3.1, the metrics-and-structure data mining is done using third-party C++ parsers. The metric lens and metric legend visualizations described in

Sec. 4.1 and 4.2 are implemented using OpenGL atop of an existing UML diagram visualizer [35]. 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[1], 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 methods, and class frame widths, based on the method signature lengths[3]. Class member signatures are available from the architecture extraction phase. 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 [13, 9], SUGIBIB [11, 12], Tom Sawyer Software [37], or VCG [19] tools, at the expense of a more complex implementation.

Additionally, we visualize other architectural aspects using the *areas-of-interest* (AOI) visualization technique [6]. Sets of classes in a diagram which constitute an aspect, *e.g.* all classes in a design pattern, are drawn surrounded by a fuzzy, smooth contour, constructed as explained in [6] (see *e.g.* Fig. 6).

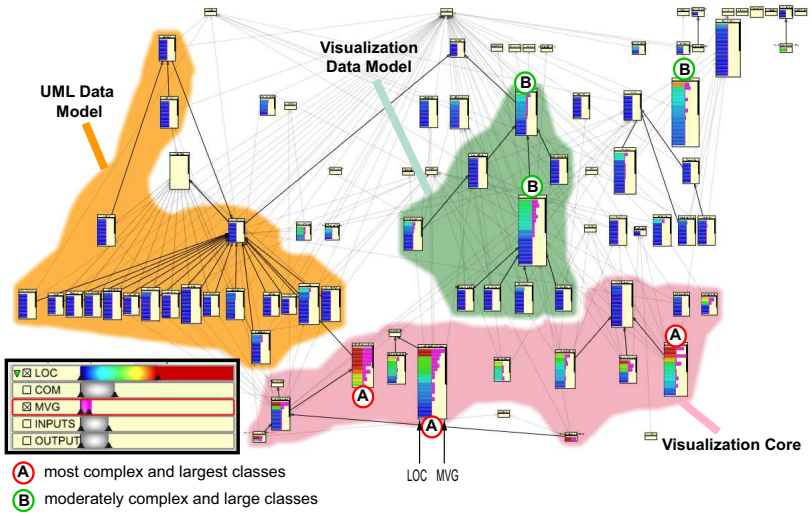

Figure 6: Complexity assessment of a UML diagram with three subsystems.

[3]Data members can also be taken into account along with methods, if desired

6 CASE STUDY

To assess the effectiveness of our combined lens-and-legend metrics visualization, we conducted a case study. The study's goal was to assess the modularity and maintainability of a given C++ code base containing about 15000 lines of code. The system, a UML editor, has been developed by four individuals over a period of four years, whose expertises ranged from undergraduate student to professional C++ game developer. The current developer, involved in the last development year, has mentioned the existence of several maintenance and evolution problems, related to the use of different coding styles, badly documented parts, and undocumented dependencies.

Our study's main question was: Would a fourth 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. 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 entire data extraction step (Sec. 3.1 took around 2 hours, including the tool's set-up and configuration. Next, the investigator performed three types of analyses: a complexity assessment (Sec. 6.1, a change propagation analysis (Sec. 6.2, and a code-level documentation review (Sec. 6.3. All analyses took under three hours, and involved chiefly the usage of our visualization tool. The actual C++ source code was investigated only for about 10-15 minutes, to check some hypotheses which were not evident from the visualization alone. Finally, the investigator (one of the authors) reported and cross-checked his findings in a discussion with the current developer.

6.1 Complexity Assesment

In the 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 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 Fig. 6). The *MVG* metric is visualized with purple bars scaled to the metric

value (the right bar-graph in the classes in Fig. 6). Next, we sorted the metric lens display decreasingly on *LOC* (from top to bottom of the class icons), and also set the visible ranges of *LOC* and *MVG* to 50 and 10, respectively. We can now quickly discover methods larger than 50 LOC and/or with a complexity above 10, which are values that we consider to indicate a "complex" method, by looking for red, respectively long purple, bars in Fig. 6).

Looking at Fig. 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. 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 Fig. 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 Fig. 6).

6.2 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 diagram as for the complexity assessment, but now we consider the number of variables read *INPUTS*, respectively written *OUTPUTS*, by each method. Metrics are sorted on decreasing *INPUTS* value, 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 result is shown in Fig 7.

We quickly see that there is no correlation between *INPUTS* and *OUTPUTS* values, but also discover some interesting outliers. The class marked A reads and writes a lot. This class is responsible for the rendering of UML model elements. Following the UML diagram, we discover it inherits from a *Visitor* interface. Looking at its method signatures, 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, we assess that the strong dependency of *UMLModelVisualizer* from the Data Model subsystem is a safe, acceptable one.

Another outlier class B reads a lot of data (high *INPUTS* metrics on most of its methods). Looking at its association relations (arrows on the UML

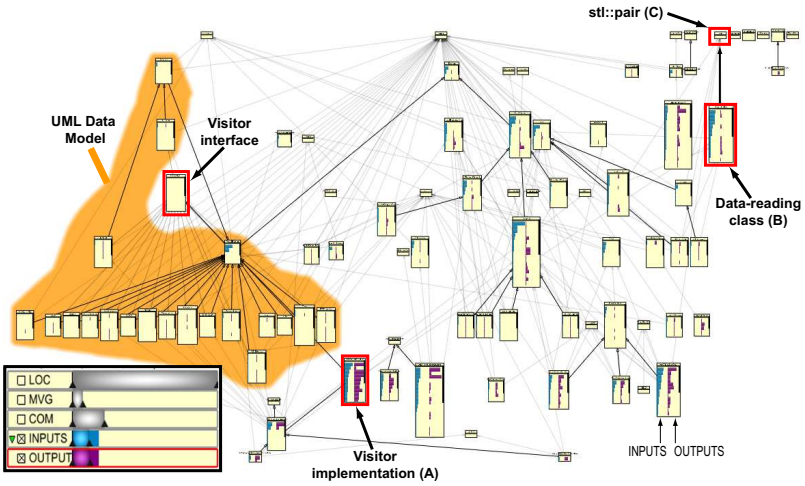

Figure 7: Change propagation analysis

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 STL C++ library. Since STL can be considered as a very stable component, 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. Figure 8 shows a zoom-in on the same UML diagram as in Figures 6 and 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 respectively, and the *LOC* metric drawn with scaled, rainbow-colored bars. If we now look at the same Visitor implementation class A, we see that it writes more data than reads. Additionally, we see that methods writing the most are also its largest methods. Given the purpose of this class, we believe that these are the methods where the core of the UML rendering activity is concentrated.

6.3 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 components are poorly commented is a typical sign of unmaintainable code.

For variation, we consider here another diagram which represents the Graphical User Interface (GUI) of the system. We display the *LOC* and comment-lines (*COM*) metrics, sorted in decreasing order of *COM*, using red, respectively blue scaled bars (Fig. 9). This will emphasize the best commented parts. Using the metric legend, we set the visible maximal values of *COM*

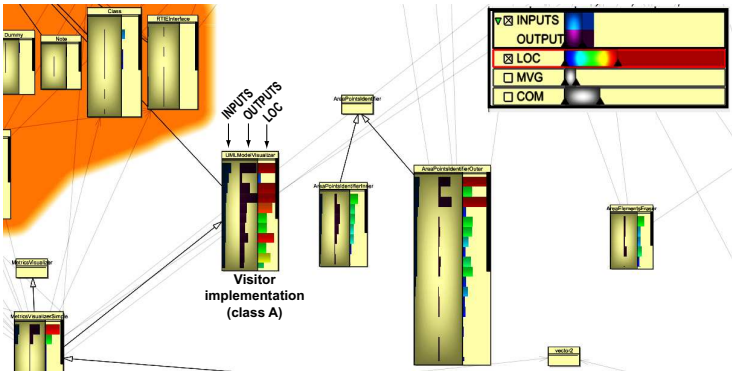

Figure 8: Correlation of number of reads and writes and method sizes

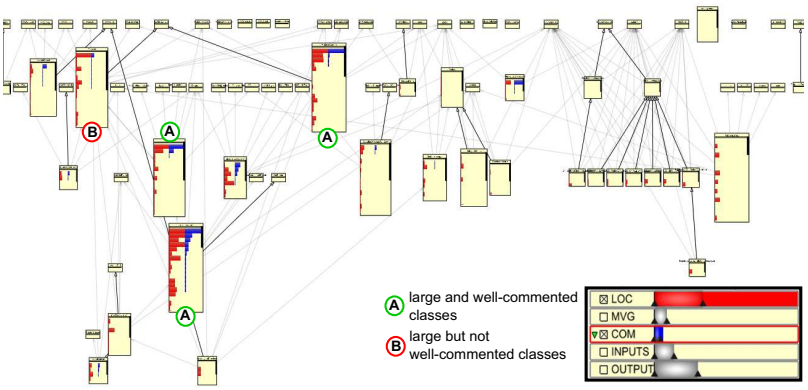

Figure 9: Code-level documentation review

and *LOC* 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 we considered to be a good comment-to-code ratio. Looking at Figure 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.

7 DISCUSSION

During validation, the developer confirmed the investigators observations and conclusions. Moreover, he also exposed the reasons for which the various outliers detected by the visualizations, *e.g.* large classes with high complexity; large and well-documented classes; and classes tightly coupled in various design patterns, occured in the design. An examples is the difference in coding experience of the involved people - the programmer who wrote the clean UML Data Model and Visitor interface was considerably more experienced than the one who wrote the Visualization Core. Also, this programmer confirmed that the largest and most complex classes are currently in an unstable state, containing mostly highly experimental code that was not reviewed. 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 by using our visualization. However, 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.

Setting up analysis scenarios with our techniques is 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. The typical analysis scenario was: load several UML diagrams, toggle through their visualizations, then focus on a particular metric or metric-structure pattern, possibly in conjunction with a given AOI, and finally zoom in at class-level (Sec. 4.1) to read the methods' names.

Our focus here was on visualizing the combined structure and metric data, and not on extracting it. Our metrics-and-structure visualization can be quite easily integrated with other reverse-engineering pipelines using different code analyzers for C++ and/or other OO 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 lens technique (Sec. 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 large diagrams (100 classes or more) makes the result overcrowded and hard to read. Increasing the class icon widths alleviates this problem, but can produce too wide diagrams which are unnatural.

8 CONCLUSIONS

We have presented an integrated approach for software visual analysis of object-oriented systems for the combined tasks of assessing maintainability and system understanding. As novel elements, we have presented two visualization techniques that enable software engineers to perform metric-metric and metric-structure visual correlations at method level on object-oriented software architecture diagrams: the metric legend and the metric lens. The metric legend enables the creation and interpretation of a wide range of visualizations, by intuitively controlling how metrics are to be visually mapped. The metric lens extends and adapts the known table lens visualization technique to compactly display several method-level metrics on UML-like diagrams. Our solution integrates structure and metric extraction from C++ source code with interactive visualizations thereof, being an example of visual analytics applied to the domain of software engineering.

Several extensions are possible. We next plan to investigate how to display more metrics on the limited space offered by a class in a software diagram, and also how to make the visual correlation of diagram relations and metrics more effective, and how to correlate metrics across the boundaries of single diagrams. Also, we plan to investigate how to visualize relations between diagrams, and how to visualize metrics defined on relations, in the same scalable way we can now visualize class and method-level metrics.

REFERENCES

[1] AT&T. The graphviz package. 2007. `www.graphviz.org`.

[2] L. A. Belady et M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.

[3] W. Bischofberger. Sniff+: A pragmatic approach to a c++ programming environment. In *Proc. USENIX C++ Conference*, pages 67–81, 1992.

[4] G. Booch. On architecture. *IEEE Software*, 23(2):16–18, 2006.

[5] Borland Inc. Together modeling toolkit. 2007. `techpubs.borland.com/together`.

[6] H. Byelas et A. Telea. Visualization of areas of interest in software architecture diagrams. In *Proc. ACM SoftVis*, pages 105–114, 2006.

[7] N. Churcher, L. Keown et W. Irwin. Virtual worlds for software virtualisation. In *Proc. SoftVis*, pages 140–148. ACM, 1999.

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

[9] G. Di Battista, P. Eades, R. Tamassia et I. G. Tollis. *Graph Drawing*. Prentice Hall, 1999.

[10] S. Diehl. *Software Visualization Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2007.

[11] H. Eichelberger. Aesthetics of class diagrams. In *Proc IEEE VISSOFT*, pages 23–31. IEEE Press, 2002.

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

[13] GDT. GDTOOLKIT: A graph drawing toolkit. 2008. `www.dia.uniroma3.it/~gdt/`.

[14] M. W. Godfrey et Q. Tu. Evolution in open source software: A case study. In *Proc. Intl. Conf. On Software Maintenance (ICSM)*, pages 131–142. IEEE Press, 2000.

[15] Yiannis Kanellopoulos, Thimios Dimopulos, Christos Tjortjis et Christos Makris. Mining source code elements for comprehending object-oriented systems and evaluating their maintainability. *ACM SIGKDD Explorations Newsletter*, 8(1):33–40, 2006.

[16] T. Klemola et J. Rilling. Modelling comprehension processes in software development. In *Proc. Intl. Conf. On Cognitive Informatics (ICCI)*, pages 329–337. IEEE Press, 2000.

[17] M. Lanza. *CodeCrawler* - polymetric views in action. In *Proc. ASE*, pages 394–395, 2004.

[18] M. Lanza et R. Marinescu. *Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006.

[19] I. Lemke et G.Sander. Vcg: Visualization of compiler graphs. In *Tech. Report 12/94*. Universität des Saarlandes, Saarbrücken, Germany, 1994.

[20] C. Lewerentz et F. Simon. Metrics-based 3d visualization of large object-oriented programs. In *Proc. VISSOFT*, pages 70–78. IEEE, 2002.

[21] A. Lienhardt, A. Kuhn et O. Greevy. Rapid prototyping of visualizations using mondrian. In *Proc. VISSOFT*, pages 67–70. IEEE, 2007.

[22] T. Littlefair. C and C++ code counter. 2007. `sourceforge.net/projects/cccc`.

[23] T. Littlefair. A C++ fuzzy grammar for ANTLR. 2008. `http://www.polhode.com/pccts.html`.

[24] G. Lommerse, F. Nossin, L. Voinea et A. Telea. The *Visual Code Navigator*: An interactive toolset for source code investigation. In *Proc. InfoVis*, pages 24–31. IEEE, 2005.

[25] J. Maletic, J. Leigh et A. Marcus. Visualizing object-oriented software in virtual reality. In *Proc. IWPC*, pages 26–35. IEEE, 2001.

[26] A. Marcus, L. Fend et J. Maletic. 3D representations for software visualization. In *Proc. ACM SoftVis*, page 2736, 2003.

[27] T.J. Parr et R.W. Quong. Antlr: A predicated-ll(k) parser generator. *Software - Practice and Experience*, 25(7):789–810, 1995.

[28] R. Rao et S. Card. The table lens: Merging graphical and symbolic representations in an interactive focus+context visualization for tabular information. In *Proc. CHI*, pages 222–230. ACM, 1994.

[29] Rational Inc. Ibm rational tool. 2007. `www-306.ibm.com/software/rational`.

[30] Scientific Toolworks Inc. *Understand* for c++. 2008. `http://www.scitools.com`.

[31] T. A. Standish. An essay on software reuse. *IEEE Trans. on Software Engineering*, 10(5):494–497, 1984.

[32] M. A. Storey, K. Wong et H. A. Müller. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2-3):183–207, 2000.

[33] A. Telea, A. Maccari et C. Riva. An open toolkit for prototyping reverse engineering visualizations. In *Proc. Data Visualization (IEEE VisSym)*. IEEE Press, 2002.

[34] A. Telea et L. Voinea. SOLIDFX: An interactive reverse-engineering environment for C++. In *Proc. CSMR*, pages 320–322, 2008.

[35] M. Termeer, C. Lange, A. Telea et M. Chaudron. Visual exploration of combined architectural and metric information. In *Proc. VISSOFT*, pages 21–26. IEEE Press, 2005.

[36] James J. Thomas et Kristin A. Cook. *Illuminating the Path: The R&D Agenda for Visual Analytics*. National Visualization and Analytics Center, 2005.

[37] Tom Sawyer, Inc. The tom sawyer graph layout software suite. 2008. `http://www.tomsawyer.com`.

[38] C. Ware. *Visual Thinking for Design*. Morgan Kaufmann, 2008.

[39] R. Wettel et M. Lanza. Program comprehension through software habitability. In *Proc. ICPC*, pages 231–240. IEEE, 2007.

[40] D. Wigg. CPP_Parser: A C++ grammar for ANTLR. 2008. `http://www.antlr.org/grammar/list`.

[41] P. C. Wong et James J. Thomas. Visual analytics. *IEEE Computer Graphics and Applications*, 24(5):20–21, 2004.

[42] J. Wust. SDMETRICS: The software design metrics tool for UML. 2006. `http://www.sdmetrics.com`.