Visualization of the Static Aspects of Software: A Survey

Pierre Caserta and Olivier Zendra

Abstract—Software is usually complex and always intangible. In practice, the development and maintenance processes are time-consuming activities mainly because software complexity is difficult to manage. Graphical visualization of software has the potential to result in a better and faster understanding of its design and functionality, thus saving time and providing valuable information to improve its quality. However, visualizing software is not an easy task because of the huge amount of information comprised in the software. Furthermore, the information content increases significantly once the time dimension to visualize the evolution of the software is taken into account. Human perception of information and cognitive factors must thus be taken into account to improve the understandability of the visualization. In this paper, we survey visualization techniques, both 2D- and 3D-based, representing the static aspects of the software and its evolution. We categorize these techniques according to the issues they focus on, in order to help compare them and identify the most relevant techniques and tools for a given problem.

Index Terms—Visualization of software, software comprehension, software maintenance, human perception.

1 Introduction

Software quickly becomes very complex when its size increases, which hinders its development. The very large amount of information represented in software, at all granularity levels, especially the tremendous number of interactions between software elements, make understanding software a very difficult, lengthy, and error-prone task. This is even truer when one has not been involved in its original development.

The maintenance process is indeed known to be the most time-consuming and expensive phase of the software life cycle [54]. Most of the time spent in this maintenance process is devoted to understanding the maintained system. Hence, tools designed to help understand software can significantly reduce development time and cost [55], [56].

Moreover, software is virtual and intangible [14]. Without a visualization technique, it is thus very hard to make a clear mental representation of what a piece of software is. Basically, visualizing a piece of software amounts to drawing a picture of the software [57], because humans are better at deducing information from graphical images than numerical information [58], [59], [60]. One way to ease the understanding of the source code is to represent it through suitable abstractions and metaphors. This way, software visualization provides a more tangible view of the software [61], [62]. Experiments have shown that using visualization techniques in a software development project increases the odds of succeeding [63], [64].

- *P. Caserta is with the LORIA Laboratory, INPL Nancy University, Office C-123, 615 Rue du Jardin Botanique, CS 20101, 54603, Villers-les-Nancy Cedex, France. E-mail: Pierre.Caserta@loria.fr.*
- *O. Zendra is with the INRIA Nancy Grand-Est, Office C-124, 615 Rue du Jardin Botanique, CS 20101, 54603, Villers-les-Nancy Cedex, France. E-mail: Olivier.Zendra@inria.fr.*

Manuscript received 7 Oct. 2009; revised 19 Mar. 2010; accepted 29 July 2010; published online 20 Aug. 2010.
Recommended for acceptance by A. MacEachren.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCG-2009-10-0240.
Digital Object Identifier no. 10.1109/TVCG.2010.110.

Visual representation of software relies on the human perceptual system [65]. Using the latter effectively is important to reduce the user cognitive load [66], [67], [68]. Indeed, how people perceive and interact with a visualization tool strongly influences their understanding of the data, hence the usefulness of the system [65], [69], [70]. In this paper, we survey 2D and 3D visualizations as a whole, since both have pros and cons [71], [72]. 2D visualizations have actually been the subject of many studies and began to appear in commercial products [73], [74], but the recent trend is to explore 3D software visualizations [75], [76] despite the intrinsic 3D navigation problem [77].

Software visualization can address three distinct kinds of aspects of software (static, dynamic, and evolution) [62]. The visualization of the *static aspects of software* focuses on visualizing software as it is coded, and dealing with information that is valid for all possible executions of the software. Conversely, the visualization of the *dynamic aspects of software* provides information about a particular run of the software and helps understand program behavior. Finally, the visualization of the *evolution of the static aspects of software* adds the time dimension to the visualization of the static aspect of software. We tackle only the visualization of the static aspects of software and its evolution in this paper; dynamic aspects of software fall beyond its scope.

Over the past few years, researchers have proposed many software visualization techniques and various taxonomies have been published [78], [79], [80], [81]. Some prior works [82], [83], [84], [85] dealt with the state of the art of similar fields at the time they were written, but they do not include the most recent developments. Other more recent works [76], [86] cover only a particular subset of the domain. Conversely, our paper is an up-to-date survey on the whole domain of visualization of the static aspects of software and its evolution. We present a very wide coverage of different kinds of visualization techniques, explaining each one and giving its pros and cons.

1077-2626/11/$26.00 © 2011 IEEE Published by the IEEE Computer Society

TABLE 1
Synthetic View of Our Paper Organization and the Classification We Propose for Methods Visualizing the Static Aspects of Software

	Level	Focus	Section	Visualization Technique	Representation	References	Year
Time T Visualization	Line	Line properties	2	Seesoft	2D colored pixel	[1], [2]	1992
				Sv3d	3D colored cuboid	[3], [4]	2003
	Class	Functioning, Metrics	3	Class BluePrint	2D layers and graph	[5], [6], [7]	1999
	Architecture	Organization	4.1	Treemap	2D/3D colored nested boxes	[8], [9], [10]	1991
				Circular Treemap	2D/3D colored nested circles	[8], [11]	1991
				City/Cities	3D city metaphor	[12], [13], [14], [15]	1993
				Sunburst	2D colored radial display	[16], [17], [18]	1998
				Solar System	3D solar system metaphor	[19], [20]	2003
				Voronoi Treemap	2D colored irregular shapes	[21]	2005
		Relationships	4.2	Dependency Structure Matrix	2D table	[22], [23], [24]	1981
				UML	2D diagrams	[25]	1996
				Geon	3D geon diagrams	[26], [27], [28]	1998
				Solar System	3D solar system metaphor	[19], [20]	2003
				Landscape	3D landscape metaphor	[29], [30]	2004
				Hierarchical Edge Bundles	2D graph with bundled edges	[31]	2006
				City/Cities	3D city metaphor with edges	[32], [33], [34]	2007
				3D Clustered Graph	3D clustered graph	[35]	2007
		Metrics	4.3	Polymetric views	2D graph	[5], [36], [37]	1999
				Solar System	3D Solar system metaphor with edges	[19], [20]	2003
				UML MetricView	2D UML diagrams with charts on top	[38]	2005
				Treemap metrics	2D nested boxes with color and texture	[39]	2005
				City	3D City metaphor	[40], [41], [42], [43]	2005
				UML Area Of Interest	2D diagrams with area of interest	[44], [45]	2006
Visualizing Evolution	Line	Changes	5.1	Code Flow	cable-and-plug wiring metaphor	[46], [47]	2007
	Class		5.2	TimeLine	3D building metaphor	[48]	2008
	Archi.	Organizational Changes	5.3.1	Hierarchical Edge Bundles	2D graph with bundled edges	[49]	2008
		Metrics Evolution		Evolution Matrix	2D matrix	[50], [51]	2001
			5.3.2	RelVis	2D Kiviat diagrams and graph	[52]	2005
				City/Cities	3D city metaphor with animation	[48], [53]	2008

We categorized the visualization techniques according to their characteristics and features so as to make this paper valuable both to academia and industry. One of our goals is to help find promising new research directions and select tools to tackle a specific concrete development issue, related to the understanding of the static aspects of software and its evolution.

Visualization of the static aspects of software can be split into two main categories: visualization that gives a picture of the software at time T on the one hand and visualization that shows the evolution of software across versions on the other hand. Both pertain to the static aspects of software but the notion of time is added in the second category. In each of these categories, we consider three levels of granularity, based on the level of abstraction : source code level, middle level (package, class or method level), and architecture level [87], [88].

Our paper is organized as follows: In the first part (Sections 2-4), we consider visualization of the static aspects of software at time T. Section 2 presents visualization techniques that help visualize source code at line level. In Section 3, we discuss visualization techniques that provide insight into classes and help understand their internals. Section 4 deals with visualization techniques that focus on visualizing three different architectural aspects of software. The first (Section 4.1) looks at the overall source code's organization packages, classes, and methods. The second (Section 4.2) looks at relationships between components, be they based on inheritance or static call graph. The third (Section 4.3) looks at metrics to visualize and manage the quality of the software.

In the second part of our paper (Section 5), we present the smaller number of visualization techniques dealing with the evolution of the static aspects of software. This part of the paper is organized in a similar way to the first part. Section 5.1 thus presents visualization techniques that help visualize source code line changes across versions. In Section 5.2, we discuss visualization techniques that provide insight into how classes evolve across versions of the software. Section 5.3 deals with visualization techniques that focus on visualizing two different architectural aspects of software evolution. The first one (Section 5.3.1) visualizes the overall source code organization changes. The second one (Section 5.3.2) visualizes how software metrics evolve. Note that we are not aware of any visualization technique that focuses on visualizing how relationships change across software versions. Finally, we draw our conclusion in Section 6.

To help the reader, Table 1 provides a summary overview of the classification of the visualization methods and the section describing the method in our paper. The table also specifies the year in which each method appeared, thus helping to outline how the software visualization field has evolved.

2 Code-Line-Centered Visualization

This section deals with visualization at the source code line level (see details in Section 2 in Table 1). Nevertheless, the visualizations presented here have a higher degree of

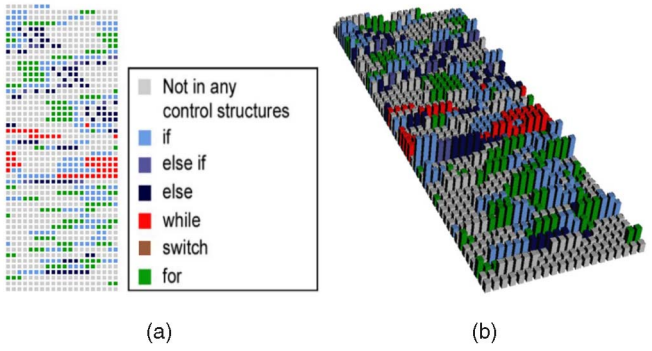

Fig. 1. (a) SeeSoft and (b) sv3D from [3].

abstraction than just displaying the source code itself. Source code editors thus fall outside the scope of this section and paper.

SeeSoft [1], [2] is a source-code-based visualization technique by Eick and coworkers. This visualization technique is a miniaturized representation of the source code lines of the software. A line of code is represented by a colored line with a height of one pixel and a length proportional to the length of the code line. Indentation of the code is preserved so the structure of the source code stays visible even when a large volume of code is displayed. To fill the window and to display large quantities of source code simultaneously, a line of code can be represented by a single colored square (Fig. 1a). This square representation reduces the visualization of the source code to a minimum, allowing the visualization of several source code files at a time by placing their representations side-by-side, thus providing rapid access to the overall source code of the software in a single view.

Marcus et al. [3], [4] based their visualization technique on SeeSoft and added a third dimension to create the *sv3D* visualization technique. The latter maps each line of source code to a *rectangular cuboid* in the same manner as the SeeSoft visualization technique. Files are represented as a group of rectangular cuboids placed on a 2D surface (Fig. 1b). To visualize the entire software, groups of rectangular cuboids (classes) are positioned in a 3D space. With this third dimension, sv3D is richer in terms of actions and interactions than SeeSoft. Users can move and rotate groups of cuboids to arrange the visualization as they wish. Zooming provides an efficient way to focus on a precise part of the visualization. The displayed information can be filtered and managed by adding transparency [89] or dispersing and regrouping rectangular cuboids of the same color at distinct heights.

The color of each pixel corresponds to a particular characteristic of the source code line. In both Figs. 1a and 1b, the color shows the type of source code line control structure. The developer is generally able to mentally re-map colored pixels to lines of code, which helps navigate through the source code. Various mappings are possible, for instance, the age of the line can be represented by a red (most recent) to blue (oldest) color scale. The creators of SeeSoft mentioned that even though individual colored squares are small, their color is perceivable and often follows a regular pattern.

In the sv3D visualization technique, the third dimension enables mapping of more source code line numerical properties. This visualization technique is very expressive because of the large number of visual parameters available to display information (height, depth, shape, x-position, y-position, and color above and below the Z axis). However, the simultaneous displaying of all the available visual parameters is likely to lead to an information overload and to fail to effectively represent the software. Balancing expressiveness and effectiveness is thus a very important criterion to obtain an understandable visualization technique [90]. In Fig. 1b, the height of the cuboid represents the nesting level of the control structure (the higher the cuboid, the higher the nesting). This mapping uses few parameters to avoid a cognitive overload.

SeeSoft was used by Bell Laboratories during the 1990s on a software containing millions of lines of code and developed by thousands of software developers. Their feedback on the Seesoft visualization technique was positive [2]. They especially appreciated being able to have both a global overview and a finely detailed view, by simply moving the mouse over a colored pixel.

Comparing the two previous visualizations shows that the SeeSoft visualization technique cannot display the control structure's nesting level at the same time as the control structure itself. sv3D, on the other hand, is able to show both the nesting level and the control structure by using the third dimension.

As far as we know, the sv3D visualization, introduced in 2003, was the last visualization of the static aspects of software to directly map source code lines to a visual representation. The more recent visualizations rely on higher level representations, more loosely associated with the source code lines.

3 Class-Centered Visualization

In this section, we present a visualization technique that helps understand the inner functioning of a class, alone or in the context of its immediate inheritance hierarchy (see details in Section 3 in Table 1).

Understanding the functioning of a single class helps grasp the way in which larger parts of the overall system work. Some research has been done on visualizing the cohesion within a class [91]. Cohesion metrics describe the degree of connectivity among software components and indicate whether a system has been well designed. However, a cohesion metric (as a numerical value) is difficult to define precisely and to quantify. Visualizing cohesion can therefore be useful to detect several design problems and give clues about the design quality.

The *class blueprint* [5], [6], [7] is a lightweight visualization technique that displays the overall structure of a class, the control flow among the methods of the class, and how methods access attributes. This visualization technique was introduced by Lanza and Ducasse.

The class blueprint (bottom left of Fig. 2) is divided into five layers (left to right): initialization, interface, implementation, accessors, and attributes. Methods and attributes are represented as nodes placed in the layer they have been assigned to. For instance, a method responsible for creating an object and initializing its attributes is placed in the first layer. Method invocation sequences are represented by

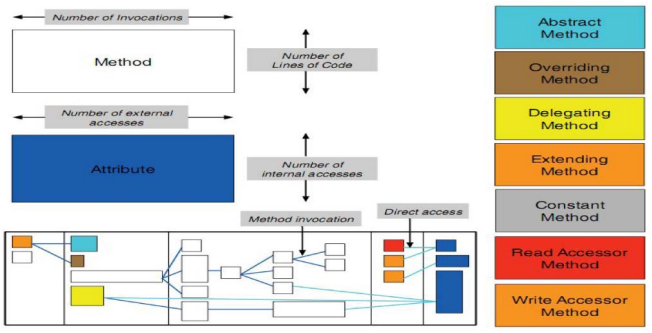

Fig. 2. Class Blueprint from [92].

edges directed from left to right. Node size is mapped to software metrics. Semantic information is mapped to the colors of nodes and links (for instance, a red node is a "getter" method).

The class blueprint visualization technique shows the purpose of methods within a class and the relationship between methods and attributes. It conveys information that is otherwise hard to notice because that would require a line-by-line understanding of the entire class. It also uses metrics to show information pertaining to methods and attributes. The metrics mapped to nodes corresponding to attributes are the number of external accesses (width) and number of internal accesses (height). The metrics mapped to nodes corresponding to methods are the number of invocations (width) and the number of lines of code (LOC) in the method (height). All these metrics except LOC are related to the coupling between modules in the software and help locate modules that require the highest maintenance effort.

With this visualization, visual patterns can be quickly detected, which improves the understanding of the class. Visual patterns use node size, distribution layers, node color, edges, and the way the attributes of a class are accessed. For example, in a class blueprint:

- several big methods can reveal an important class of the system.
- a predominant interface layer means a class that acts as an interface.
- many accessor methods (red) and many attributes (blue) with few other methods describe a class that mainly defines accessors to attributes.
- a cluster of methods structured in a deep, and often, narrow invocation tree pattern means that the developer has decomposed an implementation (probably a complex algorithm) into methods that invoke each other and, possibly, reuse some parts.
- attribute nodes that are accessed uniformly by groups of method nodes reveal a certain cohesion of the class regarding its state management since multiple methods access the same state.

More examples can be found in [5], [6], [7].

Few available visualization tools cover class understanding, like this one. Furthermore, the class blueprint visualization technique supports class understanding within the context of the immediate inheritance hierarchy, by displaying how a subclass fits within the context of its parent, brothers, and children. Visual patterns in the context of inheritance may thus be found as well. Experiments in [7] have shown that the class blueprint visualization is useful to better understand classes compared to source code reading.

4 Architecture Visualization

Visualizing the architecture of software [93] is one of the most important topics in the visualization of software field [86], [94], [95], [96]. Object-oriented software is generally structured hierarchically, with packages containing subpackages, recursively, and with classes structured by methods and attributes. Visualizing the architecture consists in visualizing the hierarchy and the relationships between software components. As is clear in Table 1, many visualization techniques tackle the architecture level.

Being able to have a global overview of the system is considered very important [97] to decide which components need further investigation and to focus on a specific part of the software without losing the overall visual context [18].

This section deals with representations of the overall architecture, such as tree, graph, and diagram model representations. We present visualization techniques focusing on visualizing three different aspects of the software architecture. First, the software global architecture, including code source organization, to see how packages, classes, and methods are organized. Second, relationships between components, be they based on inheritance or call graphs. Third, metric-centered visualizations to visualize the quality of the software. As we will detail, some representations are better to visualize an aspect of software but less efficient to visualize another.

4.1 Visualizing Software Organization

A tree model is perfect for representing packages, classes, and methods organization. But the classic node-link diagram is inappropriate for tree representation, quickly becoming too large and making poor use of the available display space (Fig. 3a). Moreover, the amount of textual information contained in nodes should be limited because too much will quickly clutter the display space, causing information overload [87], [98]. For this reason, many tree layout algorithms have been developed to optimize the visual representation of the tree [99], [100], [101], [102]. These visualizations are mainly 2D representations but they can be extended to 3D representations. This section presents several visualization techniques that show the source code organization. Most of them use tree representations that are not specific to software visualization but are applied to software (see details in Section 4.1 in Table 1, p. 2).

The *Treemap* visualization was introduced by Johnson and Shneiderman [8], [9]. It provides an overall view of the entire software hierarchy by efficiently using the display area [10] (Fig. 3b). This visualization is generated by recursively slicing a box into smaller boxes for each level of the hierarchy, using horizontal and vertical slicing alternatively. The resulting visualization displays all the elements of the hierarchy, while the paths to these elements are implicitly encoded by the Treemap nesting. Figs. 3a and 3b represent, respectively, a common tree and its equivalent Treemap (color helps understand the transformation mechanism).

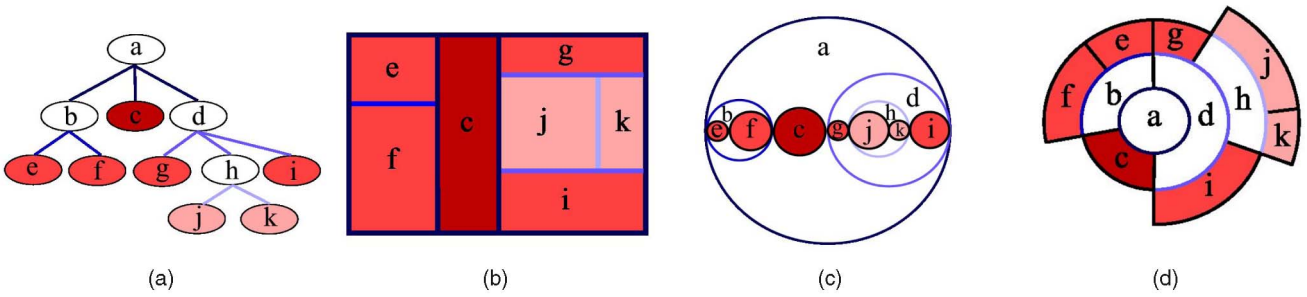

Fig. 3. (a) Node-link diagrams. (b) Treemap. (c) Circular Treemap. (d) Sunburst. These figures display the same system.

A *circular Treemap* is a visualization technique based on nested circles, where circles contain subcategories in smaller circles. Sibling nodes are at the same level and share the same parent (Fig. 3c). Representing the hierarchy using circles instead of rectangles makes it easier to see groupings and hierarchical organization, because all the edges and leaves of the original tree remain visible. Different layouts have been studied to have circles efficiently fill the available space [11].

Stasko and Zhang proposed the *Sunburst* visualization technique [18], which is an alternative to nested geometry, to represent tree models. This visualization technique relies on a circular or radial display to represent the hierarchy (Fig. 3d). It uses nested discs or portions of discs to compactly visualize each level of the hierarchy [16], [17]. Its main principle is that the deepest in the hierarchy is the furthest from the center. The smallest disc at the center of the visualization thus represents the root element, while the child nodes are drawn further from the center within the arc subtended by their parents. Experiments in [103] have shown that the performance of typical tasks (such as localization, comparison, and identification of files and directories inside huge hierarchies) with a Sunburst and a Treemap visualization is equivalent but the Sunburst is easier to learn and more pleasant than a Treemap. The size of packages, classes, and methods of the hierarchy are represented in all techniques, i.e., Treemap, circular Treemap, or Sunbust.

One drawback of the nested tree representation is that elements are hard to differentiate. Research has therefore been undertaken on a Treemap to explore the hypothesis of giving irregular shape to the elements. This is the main idea of the *Voronoi Treemap* [21] since various shapes allow a much better differentiation than mere rectangles (Fig. 4). Every package, class, and method will therefore have its own unique shape.

Another drawback of a Treemap is that the implicit hierarchical structure is hard to discern. To help distinguish the structure level in the Voronoi Treemap visualization, elements high in the hierarchy are colored dark red while objects further down the hierarchy are blurred (Fig. 4). A Shadowed Cushions Treemap [104] is another visual technique designed to provide a better insight into the internal structure of the Treemap. Experiments in [105] have shown that users prefer interacting with this shadowed cushion Treemap and are faster in performing a substructure identification.

Some works aim to represent a Treemap and a circular Treemap in three dimensions by displaying the substructure at different levels of elevation in the 3D space [106], [107], [108], [109]. A circular Treemap can be displayed in 3D by turning circles into cylinders [11] and by linking the height of each cylinder with the depth of the element in the hierarchy (Fig. 5). Experiments in [107], [108] have shown that 3D Treemaps are better than 2D Treemaps to visualize depth in the hierarchy. Like a 3D Treemap, the 3D circular Treemap provides a better perception of the depth of elements in the tree. The notion of depth of elements in the tree is linked to the height of cylinders in the 3D visualization (Fig. 5d).

Storey et al. developed the Simple Hierarchical Multiperspective (*SHriMP*) tool [110], [111], [112]. SHriMP combines several graphical high-level visualizations with textual lower level views to ease navigation within large and complex software (Fig. 6). In fact, SHriMP regroups a set of visualization techniques among which the user can easily switch depending on his/her needs. Fig. 6 shows a Treemap visualization technique to display the software organization in a single window. Colored boxes represent packages (yellow), classes (light green), methods (green and

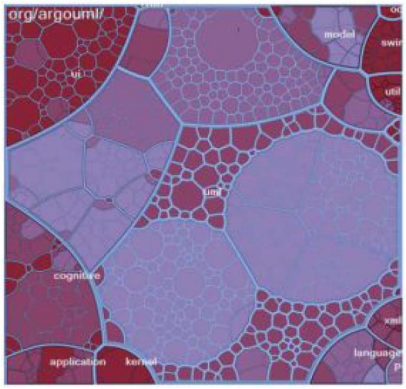

Fig. 4. Voronoi Treemap from [21].

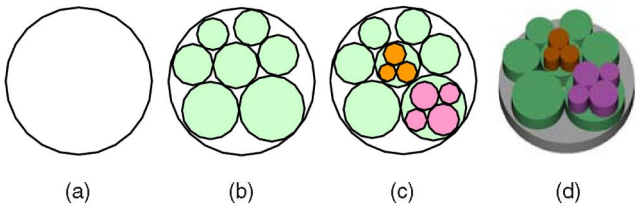

Fig. 5. (a) Level 0. (b) Level 1. (c) Level 2. (d) 3D view.

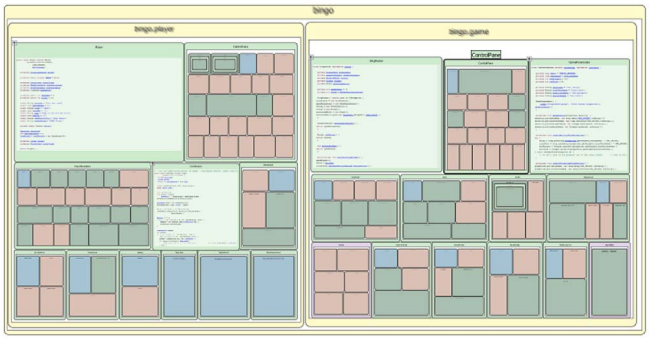

Fig. 6. The Treemap visualization from the SHriMP application.

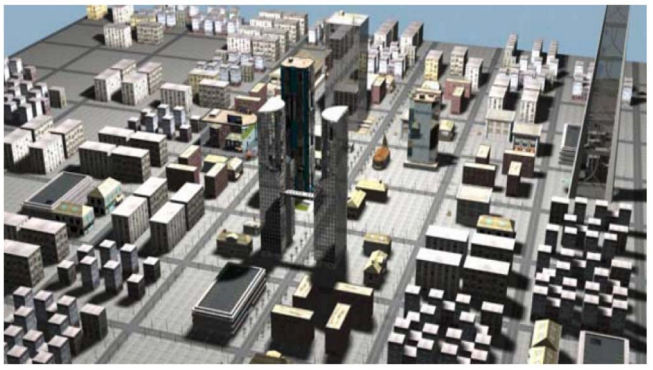

Fig. 7. Realistic City metaphor representing software from [15].

blue), and attributes (red). The most interesting feature is that each box can display its corresponding source code. This feature makes it possible to have both the source code organization and a miniaturized view of the source code itself in a single visualization. Documentation can be seen in the same way.

To ease browsing source code, SHriMP features a hypertext browsing technique on object types, attributes, and methods. The hyperlinks can be used to navigate over the source code using animated translation and zooming motions over the software visualization. The fully zoomable interface of SHriMP supports three zooming approaches: geometric, semantic, and fisheye zooming. Geometric zooming consists in scaling a specific node in the nested graph while hiding information in the rest of the system. Fisheye zooming [113] allows the user to zoom in on a particular piece of the software, while simultaneously shrinking the rest of the graph to preserve contextual information. Semantic zooming displays a particular view inside a node depending on the task at hand.

With all these features, SHriMP is a very good visualization tool to explore software structures and browse program source code. Human cognitive and perceptual capabilities are effectively exploited using smooth, interactive animations to show the part of the hierarchy focused upon. These animations allow the human perceptual system to immediately track the position of the software component focused upon [68]. This technique represents a lower cognitive burden than if the selected part is just highlighted without animated transition. In addition, according to [114], [115], it makes the visualization more understandable and more enjoyable.

SHriMP views have been very successful and integrated into several systems [116] such as the Rigi reverse engineering environment, which significantly impacted research tools and industry [117], [118], [119]. Experiments in [56] have shown that the ability to switch seamlessly between visualization techniques, as well as regrouping code, documentation, and graphical high-level views, were useful features to ease navigation and software understanding. A visualization that focuses on facilitating source code exploration can indeed enhance development and maintenance activities.

Visualizing software, using a *real-world metaphor*, consists in representing software with a familiar context, by using graphic conventions the user immediately understands [14], [89], [120], [121], [122]. According to Dos Santos et al. [123], this technique allows faster recognition of the global software structure and better and faster understanding of complex situations, preventing the most problematic aspect of 3D software visualization, i.e., disorientation. Visualization based on real-world metaphors relies on the human natural understanding of the physical world, including spatial factors in perception and navigation.

In 1993, Dieberger proposed to represent information as a *City metaphor* to solve the navigation problem [12], [13], [124]. In our paper, we use the term *City metaphor* when the whole software is represented by a single city whose buildings represent classes, whereas, we created the term *Cities metaphor* when the whole software is represented by several cities. In the Cities metaphor, the cities can represent packages (where buildings symbolize classes) or classes (where the buildings symbolize methods).

Our world, as it is structured [125], provides a natural way of splitting elements into subelements. For instance, the Earth is divided into countries, countries into cities, cities into districts, and districts contain streets, buildings, gardens, and monuments that themselves comprise buildings and gardens. These different degrees of granularity are used as an analogy for the software hierarchy. For instance, in a possible mapping, the Earth represents the overall system, countries packages, cities files, districts classes, and buildings methods. This kind of software visualization provides a large-scale understanding of the overall system. Knight and Munro [126] were the first to try representing software as Cities. They named their visualization "*The Software World*" [14], [127].

Panas et al. [15], developed a very detailed Cities metaphor to represent software. Their metaphor is as close as possible to real cities, with an extremely detailed and realistic visualization, with trees, streets, street lamps (Fig. 7). By doing this, the authors intend to enable a very intuitive interpretation of a software system. The user is free to zoom and navigate through the city representing the software.

It is strongly believed that real-world metaphors do not have to correspond exactly with reality and that small discrepancies do not hinder understanding. In fact, we consider that small discrepancies, especially simplification of the reality, can help focus on the important information of the visualization.

Graham et al. proposed another real-world metaphor to represent software: the *Solar system* [19], [20]. This

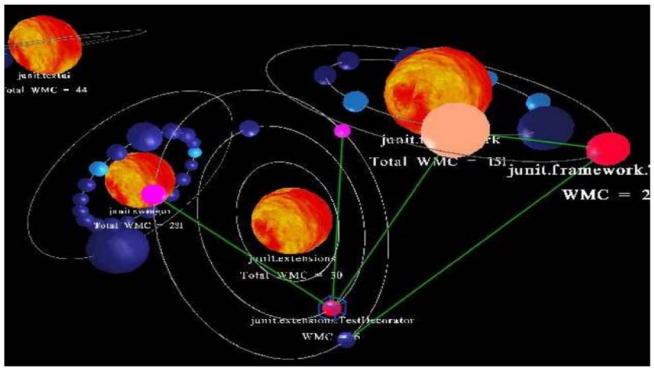

Fig. 8. Solar System metaphor from [19].

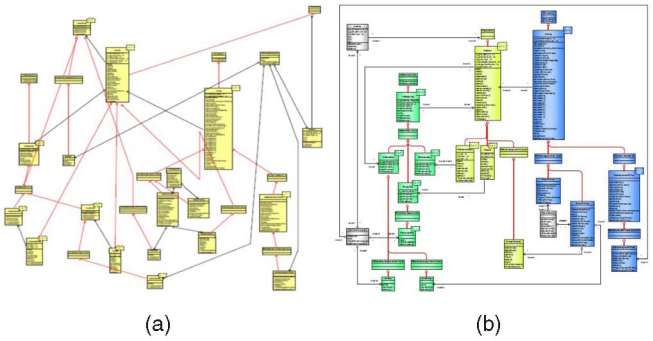

Fig. 9. (a) Classical Layout. (b) GoVisual Layout from [25].

visualization represents the software as a virtual galaxy made up of many solar systems (Fig. 8). Each solar system represents an entire package. Its central star is an icon that symbolizes the package itself, while planets, in orbit around it, represent classes within the package. The orbit level indicates the depth of a class in the inheritance tree: the farther from the star, the deeper in the inheritance tree. Blue planets represent classes, light blue planets interfaces. Solar systems are shown as a circular formation to improve readability, but the position of planets and solar systems can be moved by the user, which is a very interesting feature that few visualizations offer.

The Solar System metaphor has several interesting properties but if we compare the two real-world metaphors, it could be argued that the Solar System metaphor offers fewer levels to represent the organization of software source code. Broadening the Solar System metaphor to a Universe metaphor could, nonetheless, offer extra levels to represent packages brought together within upper-packages symbolized by Galaxies.

4.2 Visualizing Relationships in the Software

Visualizing relationships in the software is a harder task than visualizing the software hierarchy, because components can have a much larger number of relations of many kinds, such as inheritance, method calls, accesses (see details in Section 4.2 in Table 1, p. 2).

Graphs have all the characteristics needed to represent relationships between components by considering software items as nodes and relationships as edges [128], [129], [130], [131], [132]. However, visualizing all software relationships can be equivalent to visualizing a huge graph with many different interconnections, especially for large software applications. The resulting visual representation can thus be very confusing, with plenty of edge congestions, overlapping, and occlusions, which makes it almost impossible to investigate an individual node or edge.

A solution to avoid cluttered 2D graphs is 3D representation of the graphs [133], [134]. With 3D, the user can navigate to find a view without occlusions. However, navigation through a 3D graph is difficult and quickly disorientating, sometimes requiring that navigation be started again from scratch [123]. Solutions have been proposed, such as restricting the user's navigation freedom [135], or by generating automatic camera paths through the 3D graph [136].

One visual alternative to the node-link representation of graphs is a square matrix with identical row and column labels. The number of relations between a row element and a column element is shown in the matrix. Also called the *Dependency Structure Matrix* (DSM) [22], [23], [24], it provides a simple, compact, and visual representation of relations in a complex system. This technique has been successfully applied to identify software dependencies among packages and subsystems. DSM has been enriched with more visual information to identify cycles [137], [138], and class coupling [139].

UML class diagrams are probably the most popular graph-based software visualization. Their purpose is to display inter-class relations, such as inheritance, generalization, associations, aggregations, and composition. Like other graphs, when UML class diagrams grow, they become gradually visually complex and prone to information overload (Fig. 9a).

One way to reduce the visual complexity of a UML class diagrams is to reduce the number of overlapping edges. But this is not the only criterion, some studies [140], [141] point out some important esthetic preferences such as the use of an orthogonal layout (produces compact drawings with no overlap, few crossings, and few bends), the horizontal writing of the labels, the join of inheritance edges that have the same target, correctly labeled edge.

To do this, the GoVisual UML visualization takes into account esthetic preferences [25] (Fig. 9b) to provide a better perception, by drawing generalization relationships within the same class hierarchy in the same direction, avoiding nesting of class hierarchies, and using colors to highlight distinct class hierarchies and generalizations. Fig. 9b is obviously more understandable than Fig. 9a, which confirms that esthetic preferences described above are very important [142].

UML class diagrams were originally created to be drawn on 2D surfaces. Several works represent UML diagrams in 3D, using the third coordinate to display more information [26], [27], [28].

Irani et al. used *geon diagrams* to represent UML class diagrams 10(a) [143], [144]. Geons are object primitives consisting of 3D solids, such as cones, cylinders, and ellipsoids, along with information about how they are interconnected. The technique is based on an elaborate theory of structural object recognition by Biederman [59], which suggests that if the information structure can be mapped onto structured objects then the structure will be automatically

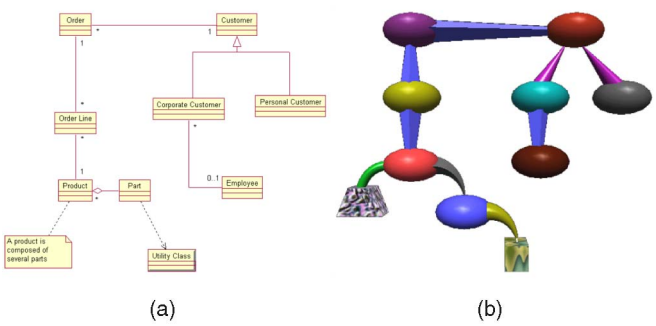

Fig. 10. (a) UML diagram. (b) Equivalent geon diagram.

extracted as part of normal perception [58]. An interesting characteristic of the geon diagram is that color and texture play a secondary role in perceptual object recognition, so they can be used to represent secondary characteristics. In Figs. 10a and 10b, we can see a UML class diagram and its equivalent geon diagram. Any UML class diagram can be turned into its equivalent geon diagram [145].

Several experiments in [146] showed that a geon diagram is both easier to understand and easier to remember than traditional UML class diagrams because of the use of simple 3D primitives.

We have already described in Section 4.1 the *SHriMP* multi-view visualization tool. SHriMP also embeds some visualization techniques to display relationships in the software.

One of them shows the static call graph of the software, representing methods by boxes and calls between methods by edges (Fig. 11). The source code of the method can thus be seen without losing the call graph context, which is very convenient to understand communications between methods. The hypertext browsing and zooming, explained in Section 4.1, are still valid for this visualization technique.

Another visualization technique, embedded in SHriMP, aims at showing every relationship in the software at once. Fig. 12 features a nested graph representing the source code hierarchy. The technique used is very similar to the SHriMP source code organization visualization described in Section 4. Relationships are directly visible as colored arcs layered over the nested graph. Arcs are labeled with the text: "extends by," "implemented by," "is type of," "calls," "accesses," "creates," "has return type," "has parameter type," "cast to type." Fig. 12 shows that a box can be turned

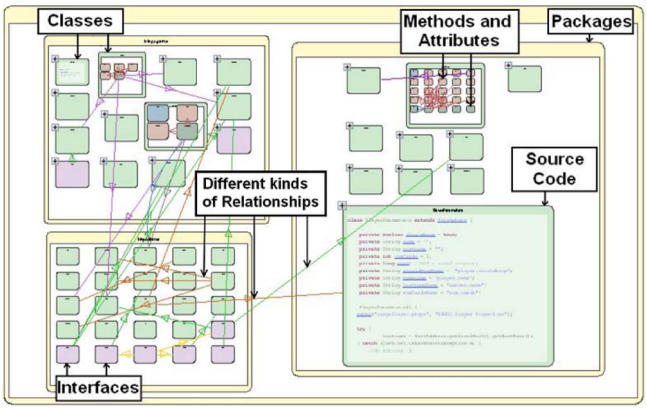

Fig. 12. Nested boxes visualization from SHriMP.

into a source code editor on demand, without loosing the high-level visualization.

Since displaying many arcs over a nested graph can lead to confusion, work has been done to reduce this visual complexity by filtering features to keep only the nodes and links that are needed to answer a specific question (e.g., display only "calls" edges that target package A)[147].

The *Extravis* tool uses a *Hierarchical Edge Bundles* visualization technique to visualize hierarchical and non-hierarchical relationships of the software created by Holten [31]. The main idea is to display connections between items on top of a hierarchical representation. Fig. 13 shows a software system and its associated static call graph. Software elements are placed on concentric circles according to their depth in the hierarchical tree. The edges are displayed above the hierarchical visualization, and represent the actual call graph. The same technique can be used on top of other visualizations of the hierarchy, such as Treemaps, circular trees, and others.

Some studies use spline edges to naturally indicate the relationship direction with no explicit arrow [148]. The Hierarchical Edge Bundles use a color interpolation on edges to represent the communication direction from caller (green) to callee (red). Thus, Fig. 13 shows that the packages at the bottom receive many calls from other packages.

To reduce the visual clutter and edge congestion, [149] bundles edge together by bending them. The edge bundling strength is controlled by a β parameter, which alters the

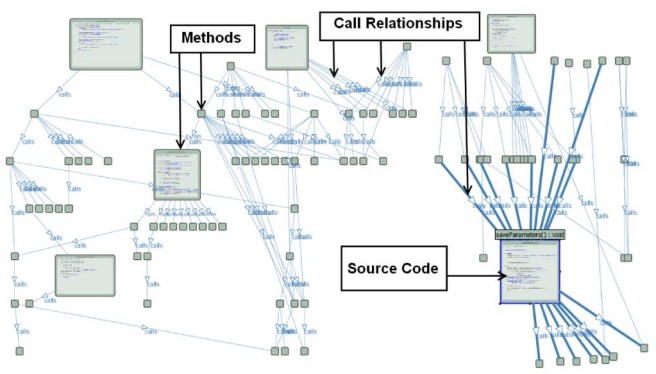

Fig. 11. Call graph visualization from the SHriMP application.

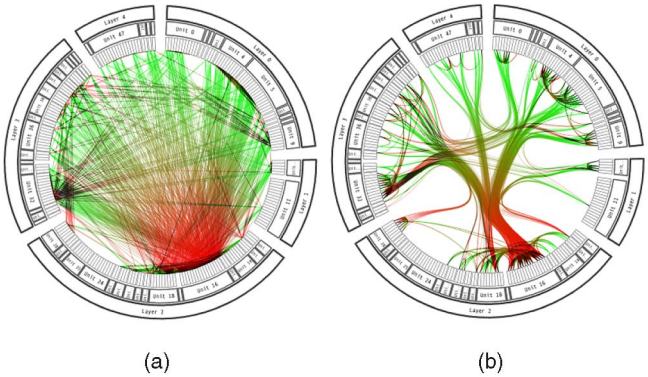

Fig. 13. Hierarchical Edge Bundles from [31]. (a) $\beta = 0$. (b) $\beta = 0.75$.

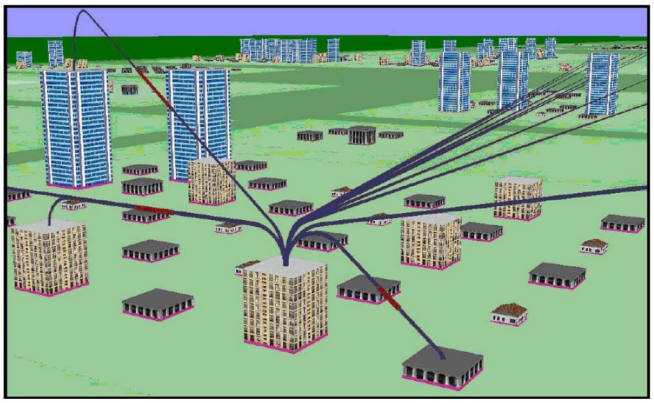

Fig. 14. Evospaces visualization from [33].

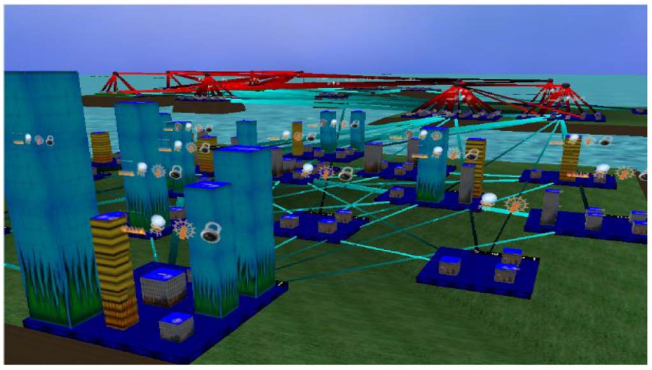

Fig. 15. The Cities and Island metaphor from [34].

shape of the spline edges (Fig. 13). Ambiguity problems are thus partially resolved, and relationships between components can be displayed in a better manner. The visualization also allows selecting a specific edge for further inspection.

To emphasize short edges, individual edges and subgroups of edges, visual alpha blending is used to draw long edges at lower opacity than short edges. This result provides an efficient rendering for visualization of edges.

Experiments in [31] show that the majority of the participants regarded the technique as useful for quickly gaining insight into class relationships. In general, the visualization was also found esthetically pleasing. Hierarchical Edge Bundles can be displayed on top of many different hierarchical structure representations but most participants preferred the radial layout over the other alternatives.

The *City or Cities metaphors* can also be used to depict relationships. The Cities visualization, presented in Section 4.1, proposes a "satellite view" to observe the cities from above. From this point of view, the user sees cities (packages) connected via streets and water. Streets represent two-directional calls between packages. Water shows unidirectional calls. Clouds cover cities that are not of current interest to the user, which provides a realistic way of filtering information.

Alam and Dugerdil proposed another way to visualize relationships with a City metaphor. Their visualization technique named *Evospaces* [32], [33] displays relationships as solid curved pipes between buildings (classes) or between objects inside buildings (methods). A colored segment moving along a pipe suggests the direction of the relation from the origin to the destination (Fig. 14). For instance, if the user wants to display call relationships of a selected class, pipes will be drawn from the selected building, and segments will move to callee classes. Curved pipes are used instead of straight ones to avoid overlapping and visual occlusions.

Textured buildings increase the feeling of navigating through a real city. This visualization provides more details when focusing on a class. The user can enter a building and see that it contains objects representing methods and local variables. Like for buildings, relationships can be displayed at method level.

Evospaces allow many interactions, such as seeing metric values, changing the appearance of objects, opening the corresponding source code file, etc. To reduce disorientation, a 2D minimap can be displayed in a corner of the screen to show the user current position in the city. This combination of 3D and 2D views helps the user make precise situation assessments for his/her navigation [150].

Panas et al. [34] use a *Cities metaphor* to represent software. We name it "*The Unified Single-View City*." In their visualization (Fig. 15), methods are represented by buildings, which are placed on blue plates that symbolize classes. These blue plates are spread on green plates that represent packages. The height of a green plate (package) depends on the depth of the package in the hierarchy, which creates virtual mountains. The resulting visualization looks like an *Island metaphor*, with sky and water between islands added for better esthetics.

Relationships between components are represented by several graphs that can be displayed alone or simultaneously. Available graphs comprise function calls, class calls, and class inheritance. Since cities are placed on plates, the resulting visualization looks like a graph with cities drawn on each node. The fact that plates are positioned on several planes, eases the display of relationships.

The visualization of Fig. 15 was created with the Vizz3d tool [151], [152]. Vizz3d helps create new representations by defining only graphical objects and the mapping, instead of hard-coding them. Vizz3D is very versatile, and the resulting visualizations can be completely different.

One drawback of City and Cities metaphors is that they can only be laid out on a two-dimensional plane that makes displaying relationships between components problematic. A solution could be walkways or roads. However, placing buildings to minimize roadways overlapping is a complicated task [19]. The Unified Single-View City tries to solve this by putting cities on several planes at different heights.

With the 3D Solar System layout presented in Section 4.1, relationships can be more efficiently displayed (Fig. 8), because every Solar System can be moved (in the 3D space) to avoid crossing edges. The Solar system visualization technique may display relationships, such as coupling and inheritance, using colored edges. This has no real-world connection but does not distort the user's understanding (Fig. 8). Nevertheless, if too many relations are displayed simultaneously, the visualization and 3D graph representation would be unclear. The authors of the 3D Solar system thought that using gravity to represent coupling seems a little too esoteric.

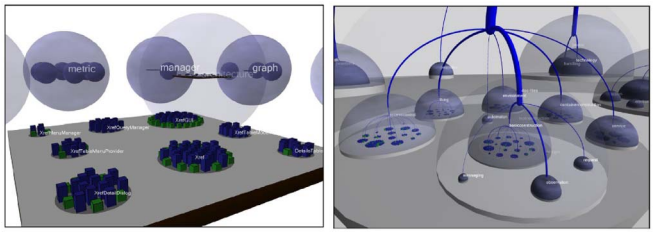

Fig. 16. (a) 3D nested spheres from [29]. (b) Hierarchical net from [30].

Balzer and Deussen proposed the *Software landscape metaphor* [29], [30] that can be considered a hybrid metaphor borrowing from the Solar system, the Island and the Cities metaphors. Its authors use 3D nested hemispheres to represent package hierarchy (Fig. 16). The outermost hemisphere represents the root package, and it contains hemispheres corresponding to the packages that are directly contained in the root package. Classes are represented as circles on a platform representing its package, while methods and attributes are displayed as simple colored boxes on the circle (Fig. 16a).

Another interesting point is how the Software landscape metaphor represents relationships between components, with a *Hierarchical net* technique. The idea is to route relationship links according to the hierarchy levels in the software (Fig. 16b). The edges of objects from a hierarchy level are grouped together and forwarded to the next (higher) level. The resulting visualization looks like a 3D tree with no overlap. This technique avoids overlapping but makes tracking a single edge difficult. Various kinds of relationships can be shown by using colors to represent the connection, whose size indicates the number of relationships.

Soon after the Software Landscape visualization, Balzer and Deussen [35] developed another 3D visualization also based on a *clustered graph layout*, to display large and complex graphs (Fig. 17). Their technique uses clustering, dynamic transparency, and edge bundling to visualize a graph without altering its structure or layout. The main idea is to group remote vertices and classes into clusters. The contents of a cluster are visible when the user focuses on a subpart of the graph, without being bothered with the details of distant clusters. The degree of transparency is dynamically adapted to the viewer location [153]. The nearer the viewer, the more transparent the cluster, with the most remote ones hiding their contents. This is a very interesting feature, because it allows a comprehensible interactive visualization of large systems with detailed information on focused parts. In addition, edges are routed and bundled together visually as one port of communication between two clusters, which makes it easier to track an individual edge. An edge shows more details when the user view is focused on it.

The visibility of nodes and links is thus changing continuously, with smooth transitions in terms of detail levels while navigating through the visualization, which makes large graphs manageable. Fig. 17 shows a graph with more than 1,500 nodes and 1,800 edges within 126 clusters, representing the inheritance relationships between classes of a large piece of software.

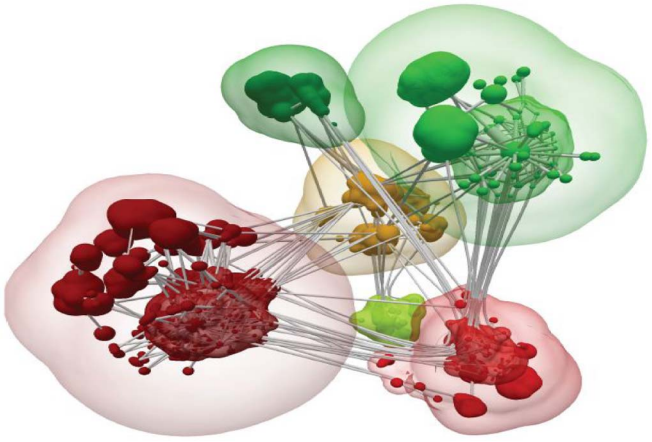

Fig. 17. Clustered graph layout from [35].

The clustered graph layout produces a more comprehensible representation of complex graphs. The example in Fig. 17 shows inheritance relationships, but another interesting use of this technique is to represent coupling between classes. Since coupled classes are likely to be considered together during development, these classes are placed close to one another in the 3D space, so as to be considered as a big cluster.

4.3 Metric-Centered Visualization

A software metric is a numeric measure of some property of a piece of software or its specifications [154], [155], [156], [157], [158]. Its purpose is to quantify a particular characteristic of a software. Software metrics are interesting because they provide information about the quality of the software design [159], [160], [161] and provide ways to monitor this quality throughout the development process [162]. Static software metrics effectively describe various aspects of complex systems, e.g., system stability, resource usage, and design complexity.

The visualization of software can transform the numerical data provided by metrics into a visual form that enables users to conceptualize and better understand the information [70], [163], [164]. The main challenge is to find effective mappings from software metrics to graphical representations [165].

In this section, we describe several visualization techniques that use static software metrics. Most of these visualization techniques use an existing visualization but properties on graphical elements are related to software metrics. For instance, a software metric related to the size of the software component can be mapped to the size of the graphical elements (see details in Section 4.3 in Table 1, p. 2).

Software metrics can effectively describe various aspects of complex architectures, and are thus worth combining with UML class diagrams. UML class diagrams provide a representation of the software design but they show little information. Termeer et al. [38] proposed the *MetricView* visualization that displays metric icons on top of UML diagram elements (Fig. 18) [36]. Bar and pie charts are mapped to metric values, in a configurable way. A 3D mode of this visualization is available just by turning 2D bars in 3D cuboids.

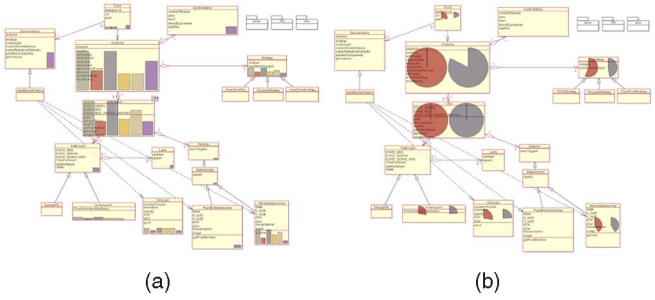

Fig. 18. (a) MetricView height bars. (b) MetricView pie charts.

The main drawback of this technique is the occlusion of the diagram by the icons, especially with 3D icons. This can be partially solved by using sliders to control the transparency of the UML diagram and the metric icons.

Instead of drawing icons on top of UML elements, Byelas and Telea [44], [45] developed a method named *area of interest* (AOI) that brings together software elements that share some common properties (Fig. 19) by building a contour that encloses these elements, and by adding colors within the AOI to display software metrics. The AOIs are shown on top of the UML class diagrams without changing its layout, to prevent disorientation.

Each AOI has its own texture such as horizontal lines, vertical lines, diagonal lines, and circles. These simple textures differentiate better when they overlap, creating a visually different pattern. Visual techniques, such as shading and transparency, are also used for a better perception of several areas, thus minimizing the visual clutter. One different metric can be associated to each AOI and its value is added to the texture using a red (high) to blue (low) color scale. This helps to show how metric values change over one AOI. When AOIs overlap, the color information for each AOI still can be seen on the texture.

The system shown in Fig. 19, represents over 50 classes with seven areas of interest. Here, the AOIs regroup classes that have a high level functional property in common, for instance classes containing the user interface code (A1: GUI), classes that are the application entry point (A5: main), and classes containing the application control code (A6: core). Byelas and Telea defined the participation degree $p_{i,j}$ of a class C_j in an area A_i as the code percentage of C_j specific to A_i: an OpenGL class has $p = 0.5$ if it has 50 percent OpenGL-specific code. The authors consider that their participation degree metric helps understand how the identified property maps to actual classes, i.e., whether the code follows the intended design, and whether there are modularity problems.

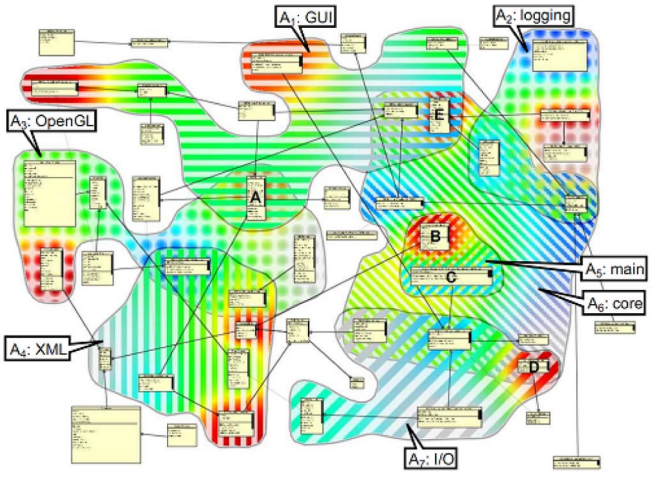

Fig. 19. Areas Of Interest from [45].

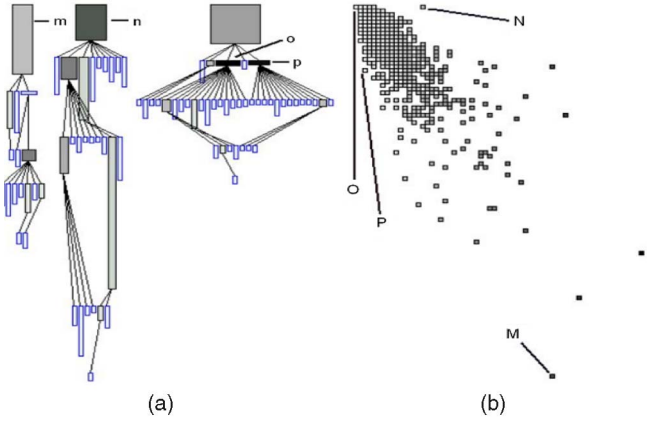

Fig. 20. Polymetric views from [36]: (a) Inheritance Tree. (b) Correlation graph.

Several facts can be noticed from Fig. 19:

- Few AOI overlap, so few classes participate in two high level functional properties, and none takes part in three, which indicates a good functional modularity.
- The B class is over a red color in A5 and A6, so the class is strongly involved in two different AOIs.
- The E class is over a red color in A6 and a blue color in A1, so the class participates strongly in the former and weakly in the latter.

Authors have mentioned that if more than three AOIs overlap then information becomes hard to understand. Nevertheless their visualization can show up to 10 areas of interest, each with its own metric, on a diagram of up to 80 classes. However, we think this visualization may be prone to eyestrain because of the large number of colors and textures.

Polymetric views [5], [36], [37], created by Lanza et al., is a lightweight visualization of software enriched with software metrics (Fig. 20). They implemented this technique in the "*CodeCrawler*" tool [166] built on top of the Moose reengineering framework [167], [168]. This visualization technique is also available as a stand-alone tool and as an an open-source software visualization plug-in for the Eclipse framework. Polymetric views propose a completely configurable graph-based representation to create multiple visualizations, depending entirely on the mapping between metrics and visual characteristics. The authors wanted to have a very clear definition of each metric, so only *direct* ones are available, which means that their computation does not depend upon another metric [169].

All the graph characteristics are based on metrics that define node width, height, x and y-coordinates and color, and edge width and color. This freedom of mapping makes it possible to visualize several aspects of the software, using the most appropriate visualization. For instance, Fig. 20a shows

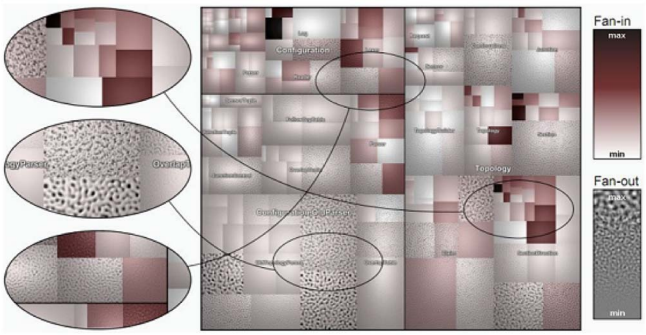

Fig. 21. Treemap with color and texture from [39].

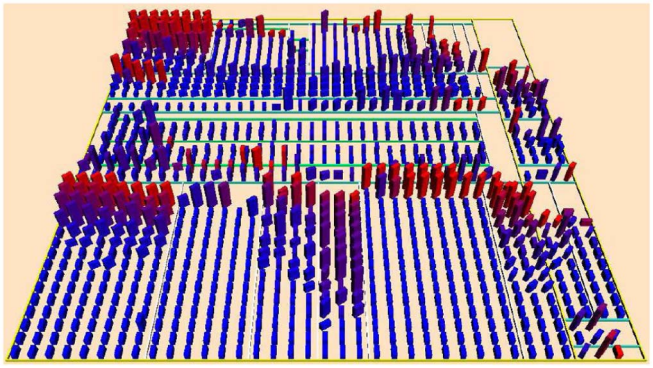

Fig. 22. VERSO from [40].

an inheritance tree where nodes represent classes and edges represent the inheritance relationship between them. Node width and height represent the number of all descendants of the class and the number of methods in the class, respectively. Node color is used to represent the number of immediate children. The goal of this mapping is to show how classes share methods through the inheritance hierarchy. Fig. 20a shows that m and n seem to be well designed because every subclass is higher than it is wide, which means they inherit parent methods and have not many immediate children. On the contrary, o and p are small, wide, and black, because they have few methods and many immediate children that are at the end of the inheritance hierarchy.

In Fig. 20b, nodes represent methods, while both node color and x-coordinate represent the number of code lines. The y-coordinate node represents the number of statements. This mapping provides a method size overview, to detect empty methods and very big methods. Fig. 20b shows that method O does not include any statement, and method N contains several lines of code but no statement at all. These methods have to be checked further and maybe deleted. In contrast, method P has more statements than lines of code, which indicates bad code formatting. Class M, that has many statements and lines of code, seems to be a good candidate for a split.

Other visualization techniques, such as histogram, checker, staple, confrontation, circle, and others, can be designed in a similar way [5], [36], [37].

Holten et al. used texture and color to show two software metrics on a Treemap (Fig. 21) [39]. This choice is based on the fact that human visual perception enables rapid processing of texture [170]. The authors used the color range to visualize one software metric and a synthesized texture distortion to visualize another one. The use of these two visual characteristics provides a high information density, helps find correlation between metrics, and can reveal interesting patterns and potential problem areas.

In Fig. 21, methods are the deepest level of the hierarchy and are visualized as boxes, while packages and classes are implicitly represented (see Section 4.1). This mapping is composed of two metrics:

- the number of callers (FANIN) is shown by a color scale white-pink-red-black.
- the number of callees (FANOUT) is represented by a texture distortion scale. The higher the metric, the more distorted the texture.

In the object-oriented paradigm, it is generally desirable that methods with high FANIN (critical methods) have low FANOUT to keep them from depending on too many other methods. The first ellipsis from the top shows methods with high FANIN and low FANOUT. The second ellipsis shows methods with low FANIN and high FANOUT. The third shows methods with fairly high FANIN and medium FANOUT, which may reveal a design issue.

Langelier et al. [40] think that the simplicity of the chosen visual representation is crucial for human perception. Their visualization, named "*VERSO*," uses simple boxes to represent classes, and cylinders for interfaces (Fig. 22). The human visual system is a pattern seeker of enormous power and subtlety [164], and relying on these simple shapes fosters a better and faster recognition of the underlying information without overloading the visualization.

Three graphical properties of the boxes (height, color, and angle) can be used to represent metrics. Although any metric can be mapped to any of the properties, the default mappings have the following underlying semantics:

- Box height is naturally related to code size.
- Box color is related to coupling. Since red is associated with danger, red symbolizes high coupling, which is dangerous in the object-oriented paradigm.
- Box angle represents the lack of cohesion, because twisted boxes look more chaotic.

The software architecture is shown either by a Treemap or a sunburst layout (Section 4) representing the package hierarchy, while classes are represented by boxes within their package.

The inheritance hierarchy is not represented but the user can select a class and see classes that have an inheritance relationship with it. Several types of relationships are available such as "in," "out," and "in/out," "aggregation," "generalization," "implementation," "invocation." The visual importance of useless elements can be reduced to focus on a subset of elements. A global view of the system is kept by only modifying the colors of filtered boxes. Filters are based on metric values or inheritance relationships.

This visualization places the human at the center of the decision process to help semiautomatically detect anomalies that are tricky to fully automatically detect [41]. One example is the detection of a Blob antipattern [171], which requires checking numerous criteria. Indeed a Blob class is a

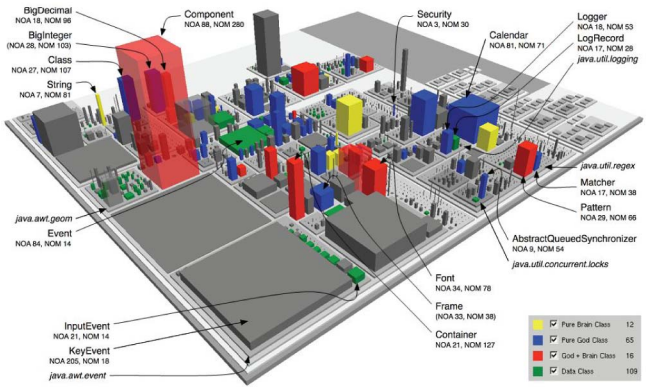

Fig. 23. CodeCity class-level disharmonies map from [43].

class that has a high complexity, low cohesion, and whose classes related by an "out" association have little complexity, high cohesion and are deep in the inheritance tree. To help detect a Blob, complexity is first mapped to height, lack of cohesion is mapped to twist, and depth in the inheritance tree is mapped to color. A filter is then applied to select only the classes with a very high level of complexity (to reduce the search space, classes with an extremely high value are colored in black). Then twisted classes are checked to identify potential suspects. Finally, a filter is applied on each potential Blob to see if associated "out" classes are small (low complexity), straight (high cohesion), and blue (deep in the inheritance tree). If so, the selected class is indeed a Blob antipattern.

The process to find design anomalies may be a bit complex but Dhambri et al. point out some important facts about automatic design anomaly detection [41]. First, there is no consensus on how to decide whether a particular design violates a quality heuristic. Second, detecting dozens of anomaly candidates is not helpful, if they contain many false positives. The authors said that the detection is more effective and useful when it is seen as an inspection task supported by a software visualization tool. We agree. Experiments in [41] confirmed that using VERSO to find anomalies is easier than manual inspection, and that the variability of the number of anomalies detected is lower when using VERSO than when conducting a full manual search.

CodeCity is a visualization tool proposed by Wettel and Lanza that represents the software with a *City metaphor* [42]. This visualization is also available as a stand-alone tool and as an an open-source 3D software visualization plug-in for Eclipse. CodeCity displays classes as buildings, and packages as city districts. Classes within the same package are grouped together, using a Treemap layout to slice the city into districts (Fig. 23).

Another idea to define building location in the city is proposed in [15]. The authors chose the amount of coupling between software components to determine building layout: the closer the buildings, the higher the coupling. This layout is interesting, because seeing coupling this way is very intuitive and very efficient for detecting problematic areas within the software. But the overall layout may drastically change between versions, which confuses the user [172].

In CodeCity, many metrics can be mapped to city characteristics. The default mapping creates an intuitive semantic mapping that helps reason about the system. The Number Of Methods (NOM) metric is mapped to building height, thus showing the amount of functionality in classes, while the Number Of Attributes (NOA) metric is represented as the width of a building. A class with many methods, but few attributes, is thus represented as a tall and thin building, whereas a class with a large number of attributes and few methods appears as a platform. The authors claim that this visualization reveals big and thin class extremes, and also gives a feeling of the magnitude of the system and the distribution of the system intelligence.

However, the resulting city looks unrealistic because very diverse building shapes appear. This is problematic because it goes against the gestalt principles [173] that state that humans efficiently distinguish at most six different shapes. The authors [174] think that having only five different building heights can reduce the cognitive load, because humans tend to organize visual elements into groups or patterns when objects look alike [175]. Moreover, using only five different heights of building decreases the visual complexity and makes a more realistic and familiar city, hence easing navigation. In the EvoSpace visualization (Section 4.2), the authors push this principle a bit further by accentuating the differences with textures on buildings. The textures used from the smallest kind of building to the highest are: house texture, city hall texture, apartment block texture, and office buildings texture (Fig. 14).

The color and transparency of several buildings can be changed by the user to visually focus on a subset of buildings. The point of view can be moved around the visualization with a high degree of freedom, but to limit disorientation, it is not possible to pass through buildings or go underground.

The CodeCity visualization helps detect high-level design problems, using software metrics [43]. Classes with design issues are singled out by coloring their representative building with a vivid color corresponding to the issue. This creates a visual "disharmonies map" of the city, providing a way to focus on problematic classes (Fig. 23).

CodeCity also features a finer-grained representation that extends the previous one by explicitly depicting the methods [43]. Each building is now composed of a set of bricks representing methods within the class. The disharmonies map can be shown at the method-level, and thus provide a very precise localization of design problems (Fig. 24).

Other techniques to spot potential design issues have been proposed. In the realistic City metaphor presented in Section 4.1, the authors of [15] mapped building texture to a metric related to the quality of the class source code. Old and collapsed buildings indicate source code that needs to be refactored, which is a realistic and intuitive way to indicate design problems. A second technique, the "*CocoViz*" visualization proposed by Boccuzzo and Gall [176], [177], focuses on visualizing whether a system is well-designed or not. Their main idea is that misshaped objects symbolize potentially badly designed software elements. In addition, misshaped objects draw attention; so they are very easy to spot. Simplified 3D real objects represent classes. For instance, the house metaphor is

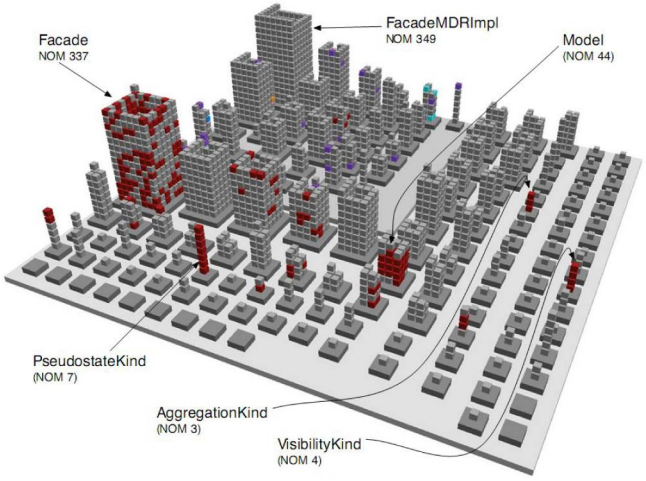

Fig. 24. CodeCity method-level disharmonies map from [43].

composed of a cone for the roof and a cylinder for the body. Each visual characteristic of these two objects may represent a normalized software metric. If the house is out of proportion, then the class may have a design problem.

In the Cities and Island metaphor presented in Section 4.2, the authors claimed that showing a lot of information in a single view was better than in multiple views [67], because a single view is easier to navigate through and always presents the same familiar picture of the system. Their visualization thus displays a lot of information (architecture, relationships, metrics) through a unique *unified single-view visualization* [34] (Fig. 15).

As all available metrics have to be displayed in one single view, the authors add 2D icons on top of each building to convey information. The height, width, depth, and texture of every visual object can express metrics as well. This visualization technique thus provides a very complete summary of the overall system but the underlying problems of single view are information overload and disorientation [67]. The key asset of this visualization, in order to overcome information overload, is the use of the cities metaphor to represent the structure of the system architecture. However, no empirical experiments were conducted on this visualization technique.

Nevertheless, we agree that the city metaphor is adapted to implicitly represent the software structure and metrics, displaying a lot of information at the same time.

On the contrary, the Solar system metaphor presented in Section 4 can only show one metric at a time, because planet size is the only parameter available to map a metric. So the visualization is split into five modes, where five different metrics are mapped to planet size: lines of code (LOC), weighted methods per class (WMC), depth of inheritance tree (DIT), number of children (NOC), and coupling. These metrics are part of the well-known CK metric set, which has been empirically validated as a good quality indicator and as a useful predictor of fault-prone classes [178], [179], [180]. In addition to mapping the chosen metric to the planet size, the metric value is also displayed as text under each planet. Filters can be applied to the overall system to display only the planets that have metric values within the defined interval.

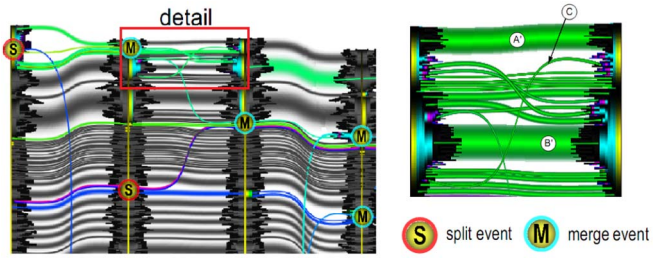

Fig. 25. Code flow from [47] (modified).

5 Visualizing Software Evolution

Visualizing software evolution is not easy because adding the time dimension implies a lot of extra data. Nevertheless, visualizing software evolution helps explain the current state of the software and depict important changes such as refactoring and growing modules [181], [182]. It provides useful information such as the modules that have been under heavy maintenance, or, on the contrary, barely touched along their entire evolution [183].

The field of software evolution visualization also includes visualizations to understand how different developers interacted on various parts of the system [184], [185], [186]. This type of visualization will not be treated in this paper because it falls beyond its scope, since we exclusively consider the software itself and not its development process.

Most techniques to visualize evolution are based on a static visualization technique, but show one "image" for each version of the software and animate the transition between them. Other types of visualization of the evolution display the entire evolution in a single image.

This section about evolution follows the same pattern as the previous parts of this paper. First, we present source code evolution visualizations in Section 5.1. Section 5.2 then tackles the visualization of single class evolution. Finally, we look into architecture evolution visualization in Section 5.3.

5.1 Visualizing Source Code Line Changes

This section presents a visualization technique to understand how the detailed structure of source code changes during development (see details in Section 5.1 in Table 1, p. 2).

Telea et al. proposed the *Code flows* visualization technique that displays source code line evolution across file versions [46], [47]. Code flows look like a *complex cable-and-plug wiring metaphor* (Fig. 25). They make it possible to visually track a given code fragment throughout its evolution with guaranteed visibility. It also highlights more complex events such as code splits and merges (S and M in Fig. 25). This level of details gives a lot of information and means that it is possible to focus on the evolution of one specific source code file. In Fig. 25, four versions of a source code class are represented from left to right, using code flows with an icicle layout [102]. Bundled edges show what a specific source code line becomes in the next version of the class. For more readability, source code lines that remain the same are colored in black.

This visualization technique provides a good general overview of a class code changes. It shows whether source code changed significantly or remained unaltered. It can display large classes with more than 6,000 lines.

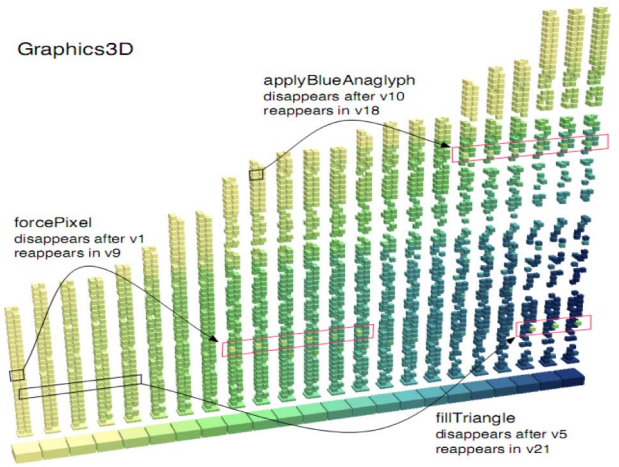

Fig. 26. Timeline from [48]: Evolution of the "Graphics3D" class of the Jmol software.

5.2 Visualizing Class Evolution

This section presents a visualization technique to understand how class methods evolve with software releases (see details in Section 5.2 in Table 1, p. 2).

Wettel and Lanza created the *timeline* visualization technique to depict class evolution (Fig. 26). It relies on a detailed *Building metaphor* where every building represents a class and every brick represents a method. Each method stays at the same spot across versions of the class; when the method disappears, the spot remains empty. Color is used to depict the age of methods, from light yellow for recent or revised methods to dark blue for the most long-standing. The age of a method is an integer value representing the number of versions that the method "survived" up to that version.

The timeline visualization is very useful to show how many methods the class defines and when new methods are created and disappear. Some evolutionary patterns can be found, for instance a building (class) that evolves and loses an ever-increasing number of bricks (methods) looks unstable. Another example is when a large number of bricks are suddenly added from one version to another, meaning that the class is becoming more important in the software. Correlating the timeline visualizations of several classes enables the detection of massive refactoring as well as files that participate in the same change [187].

5.3 Visualizing Software Architecture Evolution

Visualizing the evolution of the software architecture is one of the most important topics in the field of software evolution visualization. Having a global overview of the entire system evolution is considered very important, because it can explain and document the current state of software design.

In this section, we present visualization techniques that focus on visualizing three different aspects of the evolution. First, how the global architecture of the software, including code source organization, changes over versions. Second, how relationships between components evolve. Third, how metrics evolve with releases. Some representations are better to visualize an aspect of software but less efficient to visualize another. We will thus detail all the pros and cons of each visualization.

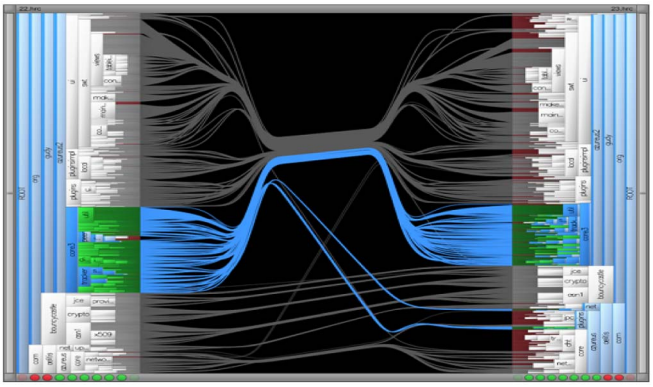

Fig. 27. Comparison of source code organization from [49].

5.3.1 Visualizing Organizational Changes

In this section, we present a visualization technique to depict changes in source code organization (see details in Section 5.3.1 in Table 1, p. 2).

Holten and Wijk proposed a technique to compare the software hierarchies of two software versions [49]. It is inspired by the visualization of source code evolution presented in Section 5.1. This visualization shows both versions of the hierarchies and how the hierarchies are related. To facilitate comparisons, both hierarchies are organized so that matching nodes are positioned opposite each other, as much as possible.

To reduce visual clutter, the Edge Bundles technique presented in Section 4.2 is used. In Fig. 27, the source code organization of Azureus v2.2 is displayed on the left (2,283 leaves), while v2.3 (3,179 leaves) is on the right. Red shading is used to depict nodes that are present in one release but not in the other. With the Edge Bundles technique, it is easier to precisely select a group of edges to highlight the selected hierarchy (Fig. 27) or directly select a node in a hierarchy.

5.3.2 Visualizing Software Metrics Evolution

Static software metrics are abstractions of the source code. They are a means to better understand, control, manage, predict, and improve software, as well as its development and maintenance processes. These metrics can reveal important characteristics pertaining to the quality of the product, thus helping develop better software during all the phases of the software life cycle. Visualizing the evolution of software metrics gives information about how the software quality evolves. In this section, we look into several visualization techniques of metric evolution across software versions (see details in Section 5.3.2 in Table 1, p. 2).

The *Evolution Matrix* is a lightweight visualization technique that displays the entire system evolution in one picture (Fig. 28) [50], [51]. In their visualization, Lanza and Ducasse combine software visualization and software metrics by using two-dimensional boxes and mapping the number of methods metric to width and the number of attributes metric to height. Each version of the software is displayed as a column, and each line represents a different class.

This visualization technique shows the evolution of the system size, additions, and removals of classes, growth and

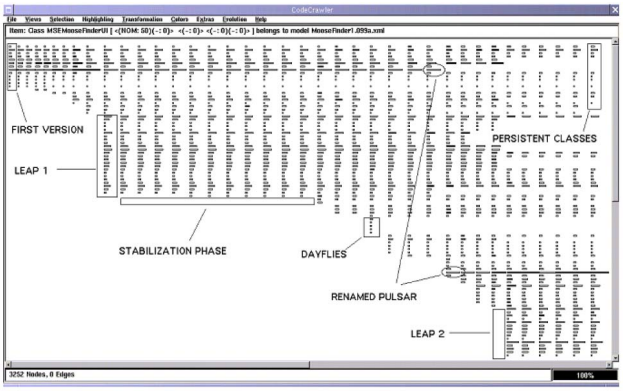

Fig. 28. Evolution Matrix from [51].

stagnation phases in the evolution. Several other patterns can be detected. For instance, classes that grow and shrink repeatedly during their lifetime can be seen as hotspots in the system; classes that suddenly grow may be the sign of unclear design or new bugs; classes that had a certain size but lost many methods may be removed.

The Evolution Matrix is thus a simple visualization approach that displays little but useful data. Its creators consider that one limitation is that the context of the inheritance hierarchy is lost with the current evolution matrix visualization. They think that introducing inheritance relationships between classes will increase the usefulness of this visualization technique.

The *VERSO* visualization by Langelier et al., presented in Section 4.3, allows the analysis of software over many versions [53]. Fig. 29a shows the appearance with VERSO of a preliminary version of a software. Fig. 29b shows the next version. A linear interpolation animation is used to go from one version to the next, the three graphical characteristics (color, height, and twist) being simultaneously incremented. The animations last only a second, but attract the visual attention of the viewer, helping understand program modifications [188].

During the visualization of the software evolution, small local changes in the software design can cause relatively large changes in the Treemap layout [189]. To solve this problem, the animation of the layout maintains all classes in the same position during the visualization of all versions, so that classes are much easier to track. However, this is a significant waste of space because of classes that have been deleted in past versions, or have not been created yet in the current version (Fig. 29). VERSO offers an alternative animation of the layout to reduce the wasted space in the early versions when only a few classes are present. It simply tries to shrink the Treemap of each version in order to use less space, but with the added constraint that all classes must keep their relative position.

Visualizing three different metrics, evolving across versions of the software, can reveal interesting patterns. Anomalies such as the God Class antipattern or the Shotgun Surgery can be detected. The former can be detected as a class, constantly growing in terms of coupling and complexity. The latter is a class constantly growing in coupling and whose complexity globally increases but with an up-and-down local pattern.

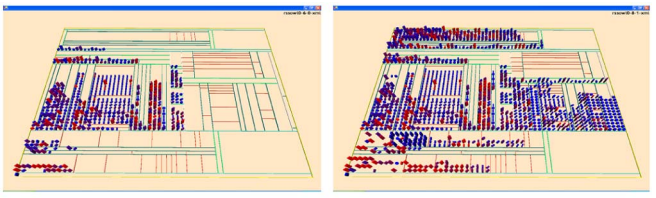

Fig. 29. Visualization of evolution with VERSO from [53].

The Wettel and Lanza *CodeCity* visualization tool, presented in Section 4.3, shows the evolution of a software by stepping back and forth through the history of the system while the city updates itself to reflect the currently displayed version [48]. The age map color mapping (defined in Section 5.2) is used to depict the age distribution of classes in the software. Fig. 30 shows the City metaphor at the method granularity level. Color ranges from light yellow for the most recent entities to dark blue for the oldest ones.

One drawback of this visualization is that, at a given time, it is impossible to know when a particular class or method disappeared, since its representation is an empty space. Furthermore, it is impossible to single out the methods whose bodies have been changed from the others. Finally, the methods of a class are hard to track when visualizing complete systems. To remedy this, Wettel and Lanza proposed the Timeline visualization technique, which focuses on one class or one package but depicts its entire evolution (Section 5.2).

Pinzger et al. proposed the *RelVis* visualization technique that can display the evolution of numerous software metrics pertaining to modules and relationships [52], [190]. This visualization technique uses a simple graph and *Kiviat diagrams* to graphically depict several metric values by plotting each value on its corresponding line (Fig. 31). Low values are placed near the center of the Kiviat, high values are far from the center. Moreover, the RelVis visualization technique shows metrics of many versions of the software by encoding the temporal order of release with a rainbow color gradient. The different colors indicate the time periods between two subsequent releases. The width of edges connecting Kiviat diagrams represents the amount of coupling between two modules. To keep the graphs understandable, relationships are drawn in the background

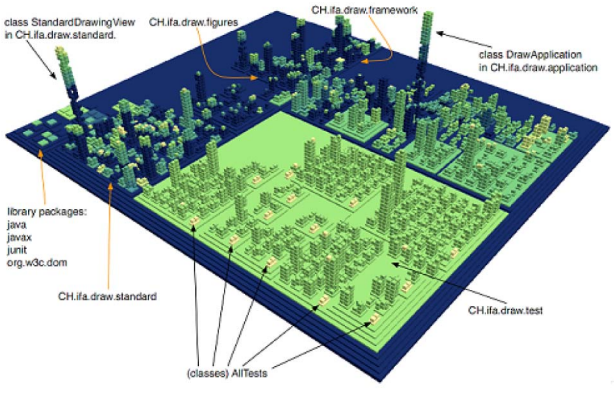

Fig. 30. Fine-grained Age Map of JHotDraw from [48].

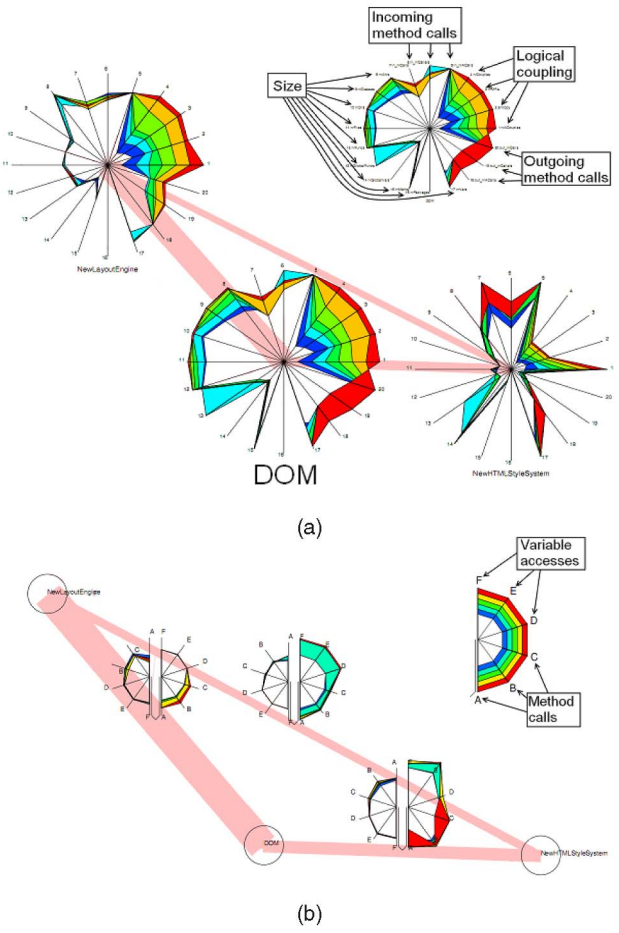

Fig. 31. RelVis from [52] (modified). (a) Metrics on modules. (b) Metrics on relationships.

with a smooth color with low contrast. The RelVis visualization technique features two representations: one to visualize a module's metrics (Fig. 31a), and the other to visualize metrics on relationships (Fig. 31b).

Fig. 31a displays 3 Kiviat diagrams representing 3 Mozilla modules. Each diagram shows 20 metrics across seven subsequent releases (numbers from 1 to 7 identify the versions). This figure is an extract of a larger one from [52]. Metrics that characterize the same property (e.g., size metrics) are grouped together. Fig. 31a highlights the `DOM` module as the largest and most complex module of the seven releases. Logical couplings increased over all seven releases. Almost all size metrics increased across versions showing that the `DOM` module constantly grew, especially from release 2 to 3 (cyan). An interesting fact is the number of global functions that dramatically increased from release 1 to 2 (blue), but decreased in the next release (cyan) and then remained constant. The number of incoming method calls increased from release 1 (blue) to the last release 7 (red) with an interesting peak from release 5 to 6 (orange). The outgoing method calls constantly increased up to release 6 (orange) but then decreased a lot from release 6 to 7 (red). Apparently, programmers resolved a reasonable amount of method call coupling.

Fig. 31b presents relationship metrics with half-Kiviat diagrams. The half-Kiviat diagram that is the closest to a node corresponds to its incoming metrics. For each relationship between source code entities, RelVis draws a straight line to indicate the relationship. Fig. 31b shows that interesting "hot-spots" are, for instance, the relationships between the module at the bottom right and the module at the top. We can notice that coupling from the module at the bottom center to the module at the bottom left decreased from release 6 to 7 (red). Apparently, release 7 tried to decouple these two modules.

This visualization contains a vast quantity of information and can be very useful to identify critical source code entities or critical couplings. However, this visualization requires a good knowledge of software metrics even though some patterns can be easily detected. A good feature of this visualization technique is the fact that all the information about metrics and evolution is displayed in one picture without using animation.

The drawback of this approach is that the colored stripes sometimes overlap. As a result, the corresponding metric values cannot be seen. An approach presented in [191] uses 3D Kiviat diagrams to display every version of the software on a different level of elevation and thus solve the problem of overlapping.

6 Conclusion

In this paper, we provide a comprehensive and up-to-date review of the literature available in the field of visualization of the static aspects of software and their evolution. The abundance of conferences, workshops, and symposia in this research field shows that it is very active and that the significance of software visualization is growing, both in academia and industry [192].

In Table 1, p. 2, we summarize the reviewed visualization techniques and categorize them according to their characteristics and features in order to help readers peruse the paper, be they academics or not.

As indicated in this paper, the software visualization domain has greatly evolved over the past few years (see the "Year" column of Table 1, p. 2), giving developers new tools to analyze, understand and develop software and helping managers to monitor the overall project to make effective decisions about its design and refactoring. However, despite the fact that software visualization tools have great potential, they are not, as yet, widespread. Substantial research is still needed to make research prototypes full-fledged tools. Even so, new commercial products recently appeared on the market, like SolidFX, an Integrated Reverse-Engineering environment for C and C++. This product includes the area of interest visualization technique and the Hierarchical Edge Bundles technique presented in this paper. Particular attention must be paid to usability issues, and usability evaluation must be improved.

A trend of this decade is to experiment with new 3D visualizations. Promising results have been obtained even though 3D navigation and interactions are problematic. They can be improved by using new technologies to interact with and display information (such as large screens, stereo views, Cave Automatic Virtual Environments, gaze tracking, motion tracking).

REFERENCES

[1] S. Eick, J. Steffen, and E. Sumner, Jr., "Seesoft—A Tool for Visualizing Line Oriented Software Statistics," *IEEE Trans. Software Eng.*, vol. 18, no. 11, pp. 957-968, Nov. 1992.

[2] T. Ball and S. Eick, "Software Visualization in the Large," *Computer*, vol. 29, no. 4, pp. 33-43, Apr. 1996.

[3] A. Marcus, L. Feng, and J. Maletic, "3D Representations for Software Visualization," *Proc. ACM Symp. Software Visualization*, p. 27, 2003.

[4] A. Marcus, L. Feng, and J. Maletic, "Comprehension of Software Analysis Data Using 3D Visualization," *Proc. 11th IEEE Int'l Workshop Program Comprehension*, pp. 105-114, 2003.

[5] M. Lanza, "Object-Oriented Reverse Engineering—Coarse-Grained, Fine-Grained, and Evolutionary Software Visualization," PhD dissertation, Univ. of Bern, 2003.

[6] M. Lanza and S. Ducasse, "A Categorization of Classes Based on the Visualization of Their Internal Structure: The Class Blueprint," *Proc. 16th ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pp. 300-311, 2001.

[7] S. Ducasse and M. Lanza, "The Class Blueprint: Visually Supporting the Understanding of Classes," *IEEE Trans. Software Eng.*, vol. 31, no. 1, pp. 75-90, Jan. 2005.

[8] B. Johnson and B. Shneiderman, "Tree-Maps: A Space-Filling Approach to the Visualization of Hierarchical Information Structures," *Proc. Second IEEE Conf. Visualization*, pp. 284-291, 1991.

[9] B. Shneiderman, "Tree Visualization with Tree-Maps: 2D Space-Filling Approach," *ACM Trans. Graphics*, vol. 11, no. 1, pp. 92-99, Jan. 1992.

[10] D. Turo and B. Johnson, "Improving the Visualization of Hierarchies with Treemaps: Design issues and Experimentation," *Proc. Third IEEE Conf. Visualizattion*, pp. 124-131, 1992.

[11] W. Wang, H. Wang, G. Dai, and H. Wang, "Visualization of Large Hierarchical Data by Circle Packing," *Proc. ACM SIGCHI Conf. Human Factors in Computing Systems*, pp. 517-520, 2006.

[12] A. Dieberger, "The Information City—a Step Towards Merging of Hypertext and Virtual Reality," *Proc. Conf. Hypertext*, vol. 93, 1993.

[13] A. Dieberger, "Navigation in Textual Virtual Environments Using a City Metaphor," PhD dissertation, Vienna Univ. of Technology, 1994.

[14] C. Knight and M. Munro, "Virtual but Visible Software," *Proc. Fourth IEEE Int'l Conf. Information Visualization*, pp. 198-205, 2000.

[15] T. Panas, R. Berrigan, and J. Grundy, "A 3D Metaphor for Software Production Visualization," *Proc. Seventh Int'l Conf. Information Visualization*, pp. 314-319, 2003.

[16] K. Andrews and H. Heidegger, "Information Slices: Visualising and Exploring Large Hierarchies Using Cascading, Semi-Circular Discs," *Proc. IEEE Symp. Information Visualization*, pp. 9-11, Oct. 1998.

[17] M. Chuah, "Dynamic Aggregation with Circular Visual Designs," *Proc. IEEE Symp. Information Visualization*, pp. 35-43, 1998.

[18] J. Stasko and E. Zhang, "Focus + Context Display and Navigation Techniques for Enhancing Radial, Space-Filling Hierarchy Visualizations," *Proc. IEEE Symp. Information Visualization*, pp. 57-65, 2000.

[19] H. Yang and H. Graham, "Software Metrics and Visualisation," technical report, Univ. of Auckland, 2003.

[20] H. Graham, H. Yang, and R. Berrigan, "A Solar System Metaphor for 3D Visualisation of Object-Oriented Software Metrics," *Proc. Australasian Symp. Information Visualization*, vol. 35, pp. 53-59, 2004.

[21] M. Balzer, O. Deussen, and C. Lewerentz, "Voronoi Treemaps for the Visualization of Software Metrics," *Proc. ACM Symp. Software Visualization*, pp. 165-172, 2005.

[22] D. Steward, "The Design Structure Matrix: A Method for Managing the Design of Complex Systems," *IEEE Trans. Eng. Management*, vol. 28, no. 3, pp. 71-74, 1981.

[23] T. Browning, L. Co, and F. Worth, "Applying the Design Structure Matrix to System Decomposition and Integration Problems: A Review and New Directions," *IEEE Trans. Eng. Management*, vol. 48, no. 3, pp. 292-306, Aug. 2002.

[24] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using Dependency Models to Manage Complex Software Architecture," *Proc. 20th Conf. Object Oriented Programming, Systems, Languages, and Applications*, 2005.

[25] C. Gutwenger, M. Jünger, K. Klein, J. Kupke, S. Leipert, and P. Mutzel, "A New Approach for Visualizing UML Class Diagrams," *Proc. ACM Symp. Software Visualization*, pp. 179-188, 2003.

[26] J. Gil and S. Kent, "Three Dimensional Software Modelling," *Proc. 20th IEEE Int'l Conf. Software Eng.*, pp. 105-114, 1998.

[27] M. Gogolla, O. Radfelder, and M. Richters, "Towards Three-Dimensional Representation and Animation of UML Diagrams," *Lecture Notes in Computer Science*, pp. 489-502, Springer, 1999.

[28] O. Radfelder and M. Gogolla, "On Better Understanding UML Diagrams through Interactive Three-Dimensional Visualization and Animation," *Proc. Conf. Advanced Visual Interfaces*, 2000.

[29] M. Balzer, A. Noack, O. Deussen, and C. Lewerentz, "Software Landscapes: Visualizing the Structure of Large Software Systems," *Proc. Joint Eurographics and IEEE TCVG Symp. Visualization*, 2004.

[30] M. Balzer and O. Deussen, "Hierarchy Based 3D Visualization of Large Software Structures," *Proc. IEEE Conf. Visualization*, p. 4, 2004.

[31] D. Holten, "Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data," *IEEE Trans. Visualization and Computer Graphics*, vol. 12, no. 5,, pp. 741-748, Sept./Oct. 2006.

[32] S. Alam and D. Ph, "EvoSpaces: 3D Visualization of Software Architecture," *Proc. Int'l Conf. Software Eng. and Knowledge Eng.*, 2007.

[33] S. Alam and P. Dugerdil, "Evospaces Visualization Tool: Exploring Software Architecture in 3D," *Proc. 14th Conf. Reverse Eng.*, pp. 269-270, 2007.

[34] T. Panas, T. Epperly, D. Quinlan, A. Saebjornsen, and R. Vuduc, "Communicating Software Architecture Using a Unified Single-View Visualization," *Proc. 12th IEEE Int'l Conf. Eng. Complex Computer Systems*, pp. 217-228, 2007.

[35] M. Balzer and O. Deussen, "Level-of-Detail Visualization of Clustered Graph Layouts," *Proc. Asia-Pacific Symp. Visualization*, 2007.

[36] S. Demeyer, S. Ducasse, and M. Lanza, "A Hybrid Reverse Engineering Approach Combining Metrics and Program Visualisation," *Proc. Sixth Working Conf. Reverse Eng.*, pp. 175-186, 1999.

[37] M. Lanza and S. Ducasse, "Polymetric Views—A Lightweight Visual Approach to Reverse Engineering," *IEEE Trans. Software Eng.*, vol. 29, no. 9, pp. 782-795, Sept. 2003.

[38] M. Termeer, C. Lange, A. Telea, and M. Chaudron, "Visual Exploration of Combined Architectural and Metric Information," *Proc. Third IEEE Int'l Workshop Visualizing Software for Understanding and Analysis*, p. 11, 2005.

[39] D. Holten, R. Vliegen, and J. van Wijk, "Visual Realism for the Visualization of Software Metrics," *Proc. Third IEEE Int'l Workshop Visualizing Software for Understanding and Analysis*, p. 12, 2005.

[40] G. Langelier, H. Sahraoui, and P. Poulin, "Visualization-Based Analysis of Quality for Large-Scale Software Systems," *Proc. 20th IEEE/ACM Int'l Conf. Automated Software Eng.*, pp. 214-223, 2005.

[41] K. Dhambri, H.A. Sahraoui, and P. Poulin, "Visual Detection of Design Anomalies," *Proc. 12th European Conf. Software Maintenance and Reeng.*, pp. 279-283, 2008.

[42] R. Wettel and M. Lanza, "Visualizing Software Systems as Cities," *Proc. Fourth IEEE Int'l Workshop Visualizing Software for Understanding and Analysis*, 2007.

[43] R. Wettel and M. Lanza, "Visually Localizing Design Problems with Disharmony Maps," *Proc. Fourth ACM Symp. Software Visualization*, 2008.

[44] H. Byelas and A. Telea, "Visualization of Areas of Interest in Software Architecture Diagrams," *Proc. Symp. Software Visualization*, 2006.

[45] H. Byelas and A. Telea, "Visualizing Metrics on Areas of Interest in Software Architecture Diagrams," *Proc. Pacific Visualization Symp.*, 2009.

[46] F. Chevalier, D. Auber, and A. Telea, "Structural Analysis and Visualization of C++ Code Evolution Using Syntax Trees," *Proc. Ninth Int'l Workshop Principles of Software Evolution*. 2007.

[47] A. Telea and D. Auber, "Code Flows: Visualizing Structural Evolution of Source Code," *Computer Graphics Forum*, vol. 27, pp. 831-838, 2008.

[48] R. Wettel and M. Lanza, "Visual Exploration of Large-Scale System Evolution," *Proc. 15th IEEE Working Conf. Reverse Eng.*, 2008.

[49] D. Holten and J. van Wijk, "Visual Comparison of Hierarchically Organized Data," *Computer Graphics Forum*, vol. 27, no. 3, pp. 759-766, 2008.

[50] M. Lanza, "The Evolution Matrix: Recovering Software Evolution Using Software Visualization Techniques," *Proc. Fourth Int'l Workshop Principles of Software Evolution,* pp. 37-42, 2001.
[51] M. Lanza and S. Ducasse, "Understanding Software Evolution Using a Combination of Software Visualization and Software Metrics," *Proc. Langages et Modeles à Objets,* pp. 135-149, 2002.
[52] M. Pinzger, H. Gall, M. Fischer, and M. Lanza, "Visualizing Multiple Evolution Metrics," *Proc. Symp. Software Visualization,* 2005.
[53] G. Langelier, H.A. Sahraoui, and P. Poulin, "Exploring the Evolution of Software Quality with Animated Visualization," *Proc. IEEE Symp. Visual Languages and Human-Centric Computing,* 2008.
[54] F. Abreu, M. Goulão, and R. Esteves, "Toward the Design Quality Evaluation of Object-Oriented Software Systems," *Proc. Fifth Int'l Conf. Software Quality,* pp. 44-57, 1995.
[55] A. Von Mayrhauser and A.M. Vans, "Identification of Dynamic Comprehension Processes During Large Scale Maintenance," *IEEE Trans. Software Eng.,* vol. 22, no. 6, pp. 424-437, June 1996.
[56] M. Storey, K. Wong, and H. Müller, "How Do Program Understanding Tools Affect How Programmers Understand Programs?" *Science of Computer Programming,* vol. 36, pp. 183-207, 2000.
[57] G. Roman and K. Cox, "Program Visualization: The Art of Mapping Programs to Pictures," *Proc. 14th ACM Int'l Conf. Software Eng.,* pp. 412-420, 1992.
[58] D. Marr, *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information.* Henry Holt and Co., 1982.
[59] I. Biederman, "Recognition-by-Components: A Theory of Human Image Understanding," *Psychological Rev.,* vol. 94, pp. 115-147, 1987.
[60] I. Spence, "Visual Psychophysics of Simple Graphical Elements," *J. Experimental Psychology: Human Perception and Performance,* vol. 16, pp. 683-692, 1990.
[61] J. Stasko, *Software Visualization: Programming as a Multimedia Experience.* MIT Press, 1998.
[62] S. Diehl, *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software.* Springer Verlag, Inc., 2007.
[63] R. Grady, *Practical Software Metrics for Project Management and Process Improvement.* Prentice-Hall, 1992.
[64] K. Zhang, *Software Visualization: From Theory to Practice.* Springer, Inc., 2003.
[65] M. Storey, F. Fracchia, and H. Müller, "Cognitive Design Elements to Support the Construction of a Mental Model During Software Exploration," *J. Systems and Software,* vol. 44, pp. 171-185, 1999.
[66] M. Scaifeand and Y. Rogers, "External Cognition: How Do Graphical Representations Work?" *Int'l J. Human-Computer Studies,* vol. 45, pp. 185-213, 1996.
[67] M. Petre, A. Blackwell, and T. Green, "Cognitive Questions in Software Visualization," *Software Visualization: Programming as a Multi-Media Experience,* pp. 453-480, MIT Press, 1998.
[68] M. Tudoreanu, "Designing Effective Program Visualization Tools for Reducing User's Cognitive Effort," *Proc. ACM Symp. Software Visualization,* 2003.
[69] M. Tory and T. Moller, "Human Factors in Visualization Research," *IEEE Trans. Visualization and Computer Graphics,* vol. 10, no. 1, pp. 72-84, Jan. 2004.
[70] S. Card, J. Mackinlay, and B. Shneiderman, *Readings in Information Visualization: Using Vision to Think,* Morgan Kaufmann, 1999.
[71] J.T. Stasko, "Three-Dimensional Computation Visualization," *Proc. IEEE Symp. Visual Languages,* pp. 100-107, 1993.
[72] S. Reiss, "An Engine for the 3D Visualization of Program Information," Technical Report CS-95-14, Brown Univ., 1995.
[73] A. Telea and L. Voinea, "An Interactive Reverse Engineering Environment for Large-Scale C++ Code," *Proc. Fourth ACM Symp. Software Visualization,* pp. 67-76, 2008.
[74] A. Telea, H. Byelas, and L. Voinea, "A Framework for Reverse Engineering Large C++ Code Bases," *Electronic Notes in Theoretical Computer Science,* vol. 233, pp. 143-159, 2009.
[75] P. Young, "Three Dimensional Information Visualisation," Technical Report 12/96, Univ. of Durham, 1996.
[76] A.R. Teyseyre and M.R. Campo, "An Overview of 3D Software Visualization," *IEEE Trans. Visualization and Computer Graphics,* vol. 15, no. 1, pp. 87-105, Jan./Feb., 2009.
[77] G. Parker, G. Franck, and C. Ware, "Visualization of Large Nested Graphs in 3D: Navigation and Interaction," *J. Visual Languages and Computing,* vol. 9, no. 3, pp. 299-317, 1998.
[78] B. Price, R. Baecker, and I. Small, "A Principled Taxonomy of Software Visualization," *J. Visual Languages and Computing,* vol. 4, no. 3, pp. 211-266, 1993.
[79] G. Roman et al., "A Taxonomy of Program Visualization Systems," *Computer,* vol. 26, no, 12, pp. 11-24, Dec. 1993.
[80] C. Knight, "System and Software Visualisation," *Handbook of Software Engineering and Knowledge Engineering,* pp. 1-17, World Scientific Publishing Company, 2001.
[81] J. Maletic, A. Marcus, and M. Collard, "A Task Oriented View of Software Visualization," *Proc. First Int'l Workshop Visualizing Software for Understanding and Analysis,* pp. 32-40, 2002.
[82] S. Bassil and R. Keller, "Software Visualization Tools: Survey and Analysis," *Proc. Ninth IEEE Int'l Workshop Program Comprehension,* pp. 7-17, May 2001.
[83] D. Gracanin, K. Matkovic, and M. Eltoweissy, "Software Visualization," *Innovations in Systems and Software Eng.,* vol. 1, no. 2, pp. 221-230, 2005.
[84] M. Storey, D. Cubranic, and D. German, "On the Use of Visualization to Support Awareness of Human Activities in Software Development: A Survey and a Framework," *Proc. ACM Symp. Software Visualization,* pp. 193-202, 2005.
[85] M. Petre and E. de Quincey, "A Gentle Overview of Software Visualisation," *PPIG News Letter,* pp. 1-10, Sept. 2006.
[86] Y. Carpendale and S. Ghanam, "A Survey Paper on Software Architecture Visualization," technical report, Univ. of Calgary, 2008.
[87] R. Koschke, "Software Visualization in Software Maintenance, Reverse Engineering, and Re-Engineering: A Research Survey," *J. Software Maintenance and Evolution: Research and Practice,* vol. 15, no. 2, Mar. 2003.
[88] T. Panas, J. Lundberg, and W. Löwe, "Reuse in Reverse Engineering," *Proc. Int'l Conf. Program Comprehension,* 2004.
[89] M. Green and J. Rekimoto, "The Information Cube: Using Transparency in 3D Information Visualization," *Proc. Third Ann. Workshop Information Technologies and Systems (WITS '93),* pp. 125-132, 1993.
[90] J. Mackinlay, "Automating the Design of Graphical Presentations of Relational Information," *Trans. Graphics,* vol. 5, no. 2, pp. 110-141, 1986.
[91] N. Churcher, W. Irwin, and R. Kriz, "Visualising Class Cohesion with Virtual Worlds," *Proc. Asia-Pacific Symp. Informattion Visualisation,* 2003.
[92] M. Lanza, R. Marinescu, and S. Ducasse, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems.* Springer-Verlag, 2006.
[93] M. Denford, T. O'Neill, and J. Leaney, "Architecture-Based Visualisation of Computer Based Systems," *Proc. Ninth IEEE Int'l Conf. Eng. of Computer-Based Systems,* pp. 139-146, 2002.
[94] A. Hatch, "Software Architecture Visualisation," PhD dissertation, Univ. of Durham, 2004.
[95] K. Gallagher, A. Hatch, and M. Munro, "A Framework for Software Architecture Visualisation Assessment," *Proc. Third IEEE Int'l Workshop Visualizing Software for Understanding and Analysis,* p. 22, 2005.
[96] K. Gallagher, A. Hatch, and M. Munro, "Software Architecture Visualization: An Evaluation Framework and Its Application," *IEEE Trans. Software Eng.,* vol. 34, no. 2, pp. 260-270, Mar./Apr., 2008.
[97] U. Wiss and D. Carr, "An Empirical Study of Task Support in 3D Information Visualizations," *Proc. Third Int'l Conf. Information Visualisation (IV '99),* 1999.
[98] A. Kerren, A. Ebert, and J. Meyer, *Human-Centered Visualization Environments.* Springer-Verlag, 2007.
[99] C. Wetherell and A. Shannon, "Tidy Drawings of Trees," *IEEE Trans. Software Eng.,* vol. SE-5, no. 5, pp. 514-520, Sept., 1979.
[100] J. Lamping and R. Rao, "The Hyperbolic Browser: A Focus + Context Technique for Visualizing Large Hierarchies," *J. Visual Languages and Computing,* vol. 7, no. 1, pp. 33-55, 1996.
[101] G. Di Battista, P. Eades, R. Tamassia, and I. Tollis, *Graph Drawing: Algorithms for the Visualization of Graphs.* Prentice Hall, 1998.
[102] T. Barlow and P. Neville, "A Comparison of 2D Visualizations of Hierarchies," *Proc. IEEE Symp. Information Visualization,* pp. 131-138, 2001.
[103] J. Stasko, R. Catrambone, M. Guzdial, and K. McDonald, "An Evaluation of Space-Filling Information Visualizations for Depicting Hierarchical Structures," *Int'l J. Human Computer Studies,* vol. 53, pp. 663-694, 2000.

[104] J. Van Wijk and H. Van de Wetering, "Cushion Treemaps: Visualization of Hierarchical Information," *Proc. IEEE Symp. Information Visualization,* pp. 73-78, 1999.
[105] P. Irani, D. Slonowsky, and P. Shajahan, "Human Perception of Structure in Shaded Space-Filling Visualizations," *Information Visualization,* vol. 5, no. 1, pp. 47-61, 2006.
[106] N. Churcher, L. Keown, and W. Irwin, "Virtual Worlds for Software Visualisation," *Proc. Software Visualisation Workshop,* 1999.
[107] F. van Ham and J. van Wijk, "Beamtrees: Compact Visualization of Large Hierarchies," *Information Visualization,* vol. 2, no. 1, pp. 31-39, 2003.
[108] T. Bladh, D. Carr, and J. Scholl, "Extending Tree-Maps to Three Dimensions: A Comparative Study," *Lecture Notes in Computer Science,* pp. 50-59, Springer, 2004.
[109] T. Bladh, D. Carr, and M. Kljun, "The Effect of Animated Transitions on User Navigation in 3D Treemaps," *Proc. Ninth Int'l Conf. Information Visualisation,* 2005.
[110] M. Storey, H. Müller, and W.K., "Manipulating and Documenting Software Structures," *Software Visualization,* pp. 244-263, World Scientific Publishing Co., 1996.
[111] J. Wu and M. Storey, "A Multi-Perspective Software Visualization Environment," *Proc. 2000 Conf. Centre for Advanced Studies on Collaborative Research,* p. 15, 2000.
[112] M. Storey, C. Best, and J. Michand, "SHriMP Views: An Interactive Environment for Exploring Java Programs," *Proc. Ninth Int'l Workshop Program Comprehension,* pp. 111-112, 2001.
[113] G. Furnas, "Generalized Fisheye Views," *ACM SIGCHI Bull.,* vol. 17, no. 4, pp. 16-23, 1986.
[114] R. Baecker and I. Small, "Animation at the Interface," *The Art of Human-Computer Interface Design,* B. Laurel, ed., pp. 251-267, Addison-Wesley, 1990.
[115] G. Robertson, S. Card, and J. Mackinlay, "Information Visualization Using 3D Interactive Animation," *Comm. ACM,* vol. 36, no. 4, pp. 57-71, 1993.
[116] M. Storey and H. Müller, "Manipulating and Documenting Software Structures Using SHriMP Views," *Proc. Int'l Conf. Software Maintenance,* pp. 275-284, 1995.
[117] H.A. Müller, "Rigi: A Model for Software System Construction, Integration, and Evolution Based on Module Interface Specifications," PhD dissertation, Rice Univ., 1986.
[118] M. Storey, K. Wong, and H. Müller, "Rigi: A Visualization Environment for Reverse Engineering," *Proc. 19th ACM Int'l Conf. Software Eng.,* pp. 606-607, 1997.
[119] H. Kienle and H. Muller, "The Rigi Reverse Engineering Environment," *Proc. Int'l Workshop Advanced Software Development Tools and Techniques (WASDeTT),* 2008.
[120] C. Dos Santos, P. Gros, P. Abel, D. Loisel, N. Trichaud, J. Paris, C. Telecom, and S. Antipolis, "Mapping Information onto 3D Virtual Worlds," *Proc. IEEE Int'l Conf. Information Visualization,* pp. 379-386, 2000.
[121] D. Ploix, "Analogical Representations of Programs," *Proc. First Int'l Workshop Visualizing Software for Understanding and Analysis,* pp. 61-69, 2002.
[122] E. Kleiberg, H. van de Wetering, and J. Van Wijk, "Botanical Visualization of Huge Hierarchies," *Proc. IEEE Symp. Information Visualization (INFOVIS '01),* 2001.
[123] C. Russo Dos Santos, P. Gros, P. Abel, D. Loisel, N. Trichaud, and J. Paris, "Metaphor-Aware 3D Navigation," *Proc. IEEE Symp. Information Visualization (INFOVIS '00),* pp. 155-165, 2000.
[124] A. Dieberger and A. Frank, "A City Metaphor to Support Navigation in Complex Information Spaces," *J. Visual Languages and Computing,* vol. 9, no. 6, pp. 597-622, 1998.
[125] K. Lynch, *The Image of the City.* MIT Press, 1960.
[126] C. Knight and M. Munro, "Comprehension with[in] Virtual Environment Visualisations," *Proc. Seventh Int'l Workshop Program Comprehension,* pp. 4-11, 1999.
[127] S. Charters, C. Knight, N. Thomas, and M. Munro, "Visualisation for Informed Decision Making; From Code to Components," *Proc. 14th Int'l Conf. Software Eng. and Knowledge Eng.,* 2002.
[128] T. Munzner, "H3: Laying Out Large Directed Graphs in 3D Hyperbolic Space," *Proc. IEEE Symp. Information Visualization (INFOVIS '97),* pp. 2-10, 1997.
[129] I. Herman, M. Marshall, and G. Melançon, "Graph Visualization and Navigation in Information Visualization: A Survey," *IEEE Trans. Visualization and Computer Graphics,* vol. 6, no. 1, pp. 24-43, 2000.
[130] C. Lewerentz and A. Noack, "CrocoCosmos—3D Visualization of Large Object-Oriented Programs," *Graph Drawing Software.* Springer-Verlag, 2003.
[131] A. Noack and C. Lewerentz, "A Space of Layout Styles for Hierarchical Graph Models of Software Systems," *Proc. ACM Symp. Software Visualization,* pp. 155-164, 2005.
[132] H. Schulz and H. Schumann, "Visualizing Graphs—A Generalized View," *Proc. IEEE Conf. Information Visualization,* pp. 166-173, 2006.
[133] M. Staples and J. Bieman, "3D Visualization of Software Structure," *Advances in Computers,* vol. 49, pp. 96-143, 1999.
[134] A. Fronk, A. Bruckhoff, and M. Kern, "3D Visualisation of Code Structures in Java Software Systems," *Proc. Symp. Software Visualization,* 2006.
[135] A. Hanson, E. Wernert, and S. Hughes, "Constrained Navigation Environments," *Proc. Scientific Visualization Conf.,* p. 95, 1997.
[136] A. Ahmed and P. Eades, "Automatic Camera Path Generation for Graph Navigation in 3D," *Proc. ACM Int'l Conf.,* pp. 27-32, 2005.
[137] A. Bergel, S. Ducasse, J. Laval, and R. Peirs, "Enhanced Dependency Structure Matrix for Moose," *Proc. Second Workshop FAMIX and Moose in Re-Eng.,* 2008.
[138] J. Laval, S. Denier, S. Ducasse, and A. Bergel, "Identifying Cycle Causes with Enriched Dependency Structural Matrix," *Proc. 16th Working Conf. Reverse Eng.,* pp. 113-122, 2009.
[139] H. Abdeen, I. Alloui, S. Ducasse, D. Pollet, M. Suen, and I. Europe, "Package Reference Fingerprint: A Rich and Compact Visualization to Understand Package Relationships," *Proc. 12th European Conf. Software Maintenance and Re-Eng.,* 2008.
[140] H. Purchase, J. Allder, and D. Carrington, "User Preference of Graph Layout Aesthetics: A UML Study," *Lecture Notes in Computer Science,* pp. 5-18, Springer, 2001.
[141] D. Sun and K. Wong, "On Evaluating the Layout of UML Class Diagrams for Program Comprehension," *Proc. 13th Int'l Workshop Program Comprehension,* pp. 317-326, 2005.
[142] M. Eiglsperger, "Automatic Layout of UML Class Diagrams: A Topology-Shape-Metrics Approach," PhD dissertation, Univ. Tübingen, 2003.
[143] P. Irani and C. Ware, "Diagrams Based on Structural Object Perception," *Proc. Working Conf. Advanced Visual Interfaces,* 2000.
[144] P. Irani, M. Tingley, and C. Ware, "Using Perceptual Syntax to Enhance Semantic Content in Diagrams," *IEEE Computer Graphics and Applications,* vol. 21, no. 5, pp. 76-85, Sept. 2001.
[145] K. Casey and C. Exton, "A Java 3D Implementation of a Geon Based Visualisation Tool for UML," *Proc. Second Int'l Conf. Principles and Practice of Programs in Java,* pp. 63-65, 2003.
[146] P. Irani and C. Ware, "Diagramming Information Structures Using 3D Perceptual Primitives," *ACM Trans. Computer-Human Interaction,* vol. 10, no. 1, pp. 1-19, 2003.
[147] M. Pinzger, K. Graefenhain, P. Knab, and H.C. Gall, "A Tool for Visual Understanding of Source Code Dependencies," *Proc. 16th Int'l Conf. Program Comprehension.* pp. 254-259, 2008.
[148] J. Fekete, D. Wang, N. Dang, A. Aris, and C. Plaisant, "Overlaying Graph Links on Treemaps," *Proc. 2003 IEEE Symp. Information Visualization,* pp. 82-83, 2003.
[149] N. Wong, S. Carpendale, and S. Greenberg, "Edgelens: An Interactive Method for Managing Edge Congestion in Graphs," *Proc. 2003 IEEE Symp. Information Visualization,* pp. 51-58, 2003.
[150] M. Tory, A. Kirkpatrick, M. Atkins, and T. Moller, "Visualization Task Performance with 2D, 3D, and Combination Displays," *IEEE Trans. Visualization amd Computer Graphics,* vol. 12, no. 1, pp. 2-13, Jan. 2006.
[151] W. Löwe and T. Panas, "Rapid Construction of Software Comprehension Tools," *Int'l J. Software Eng. and Knowledge Eng.,* vol. 15, pp. 995-1025, 2005.
[152] T. Panas, R. Lincke, and W. Löwe, "Online-Configuration of Software Visualizations with Vizz3D," *Proc. Symp. Software Visualization,* 2005.
[153] P. Young and M. Munro, "Visualising Software in Virtual Reality," *Proc. Sixth Int'l Workshop Program Comprehension,* pp. 19-26, 1998.
[154] N. Fenton, *Software Metrics: A Rigorous Approach.* Chapman & Hall, Ltd., 1991.
[155] M. Lorenz and J. Kidd, *Object-Oriented Software Metrics: A Practical Guide.* Prentice-Hall, Inc., 1994.
[156] N. Fenton and S. Pfleeger, *Software Metrics: A Rigorous and Practical Approach.* PWS Publishing Co., 1997.

[157] B. Meyer and G. EiffelSoft, "The Role of Object-Oriented Metrics," *Computer,* vol. 31, no. 11, pp. 123-127, Nov. 1998.
[158] G. Booch, R.A. Maksimchuk, M.W. Engel, B.J. Young, J. Conallen, and K.A. Houston, *Object-Oriented Analysis and Design with Applications,* third ed. Addison-Wesley Professional, 2007.
[159] D. Card and R. Glass, *Measuring Software Design Quality.* Prentice-Hall, Inc., 1990.
[160] S. Kan, *Metrics and Models in Software Quality Engineering.* Addison-Wesley Longman Publishing Co., 2002.
[161] D. Troy and S. Zweben, *Measuring the Quality of Structured Designs.* McGraw-Hill, Inc., 1993.
[162] I. Brooks, "Object-Oriented Metrics Collection and Evaluation with a Software Process," *Proc. Workshop Processes and Metrics for Object-Oriented Software Development,* 1993.
[163] C. Chen, *Information Visualization: Beyond the Horizon.* Springer, Inc., 2004.
[164] C. Ware, *Information Visualization: Perception for Design.* Morgan Kaufmann, 2004.
[165] W. Irwin and N. Churcher, "Object-Oriented Metrics: Precision Tools and Configurable Visualisations," *Proc. Ninth IEEE Int'l Software Metrics Symp.,* pp. 112-123, 2003.
[166] M. Lanza, "CodeCrawler—Lessons Learned in Building a Software Visualization Tool," *Proc. IEEE European Conf. Software Maintenance and Reeng.,* 2003.
[167] S. Ducasse, M. Lanza, and S. Tichelaar, "The Moose Reengineering Environment," *Smalltalk Chronicles,* vol. 3, no. 2, 2001.
[168] S. Ducasse, T. Gîrba, and O. Nierstrasz, "Moose: An Agile Reengineering Environment," *Proc. 10th European Software Eng. Conf. Held Jointly with 13th ACM Int'l Symp. Foundations of Software Eng.,* 2005.
[169] *IEEE Standard for a Software Quality Metrics Methodology,* technical report, Am. Nat'l Standards Inst., 1998.
[170] C.G. Healey and J.T. Enns, "Building Perceptual Textures to Visualize Multidimensional Datasets," *Proc. IEEE Conf. Visualization '98,* pp. 111-118, 1998.
[171] W. Brown, R. Malveau, H. McCormick, III, and T. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley & Sons, Inc., 1998.
[172] I. Herman, M. Delest, and G. Melancon, "Tree Visualisation and Navigation Clues for Information Visualisation," *Computer Graphics Forum,* vol. 17, pp. 153-165, 1998.
[173] D. Todorovic, "Gestalt Principles," http://www.scholarpedia.org/article/Gestalt_principles, 2010.
[174] R. Wettel and M. Lanza, "Program Comprehension through Software Habitability," *Proc. 15th IEEE Int'l Conf. Program Comprehension,* 2007.
[175] S. Few, *Show Me the Numbers: Designing Tables and Graphs to Enlighten.* Analytics Press, 2004.
[176] S. Boccuzzo and H. Gall, "Cocoviz: Towards Cognitive Software Visualizations," *Proc. Fourth IEEE Int'l Workshop Visualizing Software for Understanding and Analysis,* pp. 72-79, 2007.
[177] S. Boccuzzo and H. Gall, "Software Visualization with Audio Supported Cognitive Glyphs," *Proc. Int'l Conf. Software Maintenance,* 2008.
[178] V. Basili, L. Briand, and W. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators," *IEEE Trans. Software Eng.,* vol. 22, no. 10, pp. 751-761, Oct. 1996.
[179] R. Subramanyam and M.S. Krishnan, "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects," *IEEE Trans. Software Eng.,* vol. 29, no. 4, pp. 297-310, Apr. 2003.
[180] H.M. Olague, L.H. Etzkorn, S. Gholston, and S. Quattlebaum, "Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes," *IEEE Trans. Software Eng.,* vol. 33, no. 6,, pp. 402-419, June 2007.
[181] S. Eick, T. Graves, A. Karr, A. Mockus, and P. Schuster, "Visualizing Software Changes," *IEEE Trans. Software Eng.,* vol. 28, no. 4, pp. 396-412, Apr. 2002.
[182] A. Hindle, Z. Jiang, W. Koleilat, M. Godfrey, and R. Holt, "Yarn: Animating Software Evolution," *Proc. IEEE Int'l Workshop Visualizing Software for Understanding and Analysis,* pp. 25-26, 2008.
[183] H. Gall, M. Jazayeri, and C. Riva, "Visualizing Software Release Histories: The Use of Color and Third Dimension," *Proc. Int'l Conf. Software Maintenance,* pp. 99-108, 1999.
[184] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler, "A System for Graph-Based Visualization of the Evolution of Software," *Proc. ACM Symp. Software Visualization,* 2003.
[185] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse, "How Developers Drive Software Evolution," *Proc. Eighth Int'l Workshop Principles of Software Evolution,* pp. 113-122, 2005.
[186] X. Xie, D. Poshyvanyk, and A. Marcus, "Visualization of CVS Repository Information," *Proc. 13th Working Conf. Reverse Eng.,* 2006.
[187] M. Fischer and H. Gall, "Evograph: A Lightweight Approach to Evolutionary and Structural Analysis of Large Software Systems," *Proc. 13th Working Conf. Reverse Eng.,* pp. 179-188, 2006.
[188] M. Shanmugasundaram, P. Irani, and C. Gutwin, "Can Smooth View Transitions Facilitate Perceptual Constancy in Node-Link Diagrams?" *Proc. ACM Int'l Conf. Graphics Interface,* p. 78, 2007.
[189] Y. Tu and H.-W. Shen, "Visualizing Changes of Hierarchical Data Using Treemaps," *IEEE Trans. Visualization and Computer Graphics,* vol. 13, no. 6, pp. 1286-1293, Nov./Dec. 2007.
[190] M. Pinzger, H. Gall, and M. Jazayeri, "ArchView—Analyzing Evolutionary Aspects of Complex Software Systems," Technical Univ. of Vienna, 2005.
[191] A. Kerren and I. Jusufi, "Novel Visual Representations for Software Metrics Using 3D and Animation," *Proc. Software Eng.,* 2009.
[192] R. Bril, A. Postma, and R. Krikhaar, "Embedding Architectural Support in Industry," *Proc. Int'l Conf. Software Maintenance,* 2003.

Pierre Caserta received the master's degree in computer science in 2008 from the Université Henri Poincaré of Nancy in France. He is currently working toward the PhD degree at the LORIA Institute, Institut National Polytechnique de Lorraine, Nancy, France. He teaches at the Université de Nancy 2, IUT Charlemagne. His research interests include the visualization of software, reverse engineering, and immersive software visualization.

Olivier Zendra received the PhD degree in computer science with highest honors in 2000 from the the Université Henri Poincaré of Nancy, France. He is a tenured researcher at INRIA, France. His main research interests include the compilation and optimization of programs, memory management and automatic garbage collection, impact of the hardware-software interface on performance, and power- and energy-saving optimizations for embedded systems. He was a key designer of SmartEiffel, The GNU Eiffel Compiler.