CocoViz: Towards Cognitive Software Visualizations

Sandro Boccuzzo, Harald Gall
Department of Informatics, University of Zurich
{boccuzzo, gall}@ifi.unizh.ch

Abstract

Understanding software projects is a complex task. There is an increasing need for visualizations that improve comprehensiveness of the evolution of a software system. This paper discusses our recent work in software visualization with respect to metaphors. Our goal is to use simple and well-known graphical elements known from daily life such as houses, spears, or tables to allow a user a quick and intuitive understanding of a given visualization via their proportions. We present a software metrics configurator that handle different metaphors and allows optimizations to their graphical representation. The results so far show that large systems can be visualized effectively with metaphor glyphs, yet more case studies and more metaphor glyphs are required for a better understanding for offering a simple and cognitive visual understanding of a software system.

1 Introduction

As software evolves and becomes more and more complex, program comprehension moveover a major concern in a software project. The amount of data and the complexity of relationships between the entities are unmanageable for engineers without effective tool support.

Additionally, stakeholders are interested in different aspects of a software project. Some of them might not be interested in the entire complexity at all, but need a reflection of the state of a project (*e.g.,* project managers). Others might like to have some profound introduction about the project, while not even being allowed to access the source code (*e.g.,* auditors, potential customers). All of them can benefit from visualizations that support them in their work. There is a great opportunity in providing views on a software project to all of the different stakeholders, offering them a comprehensive status on their current software project. Software visualization aims to help the stakeholders in understanding all the gathered information about a project in aggregating the information into effective visual representations.

The main contribution of this paper is a visualization approach we call CocoViz that offers improved software comprehension compared to abstract graphical representation used in other approaches such as starglyphs [4] or parallel coordinates [9]. Furthermore we address shortcomings of the static polymetric views [13] with 1) a dynamic approach allowing to interactively filter non relevant elements; 2) a normalization approach to represent well-designed classes as well-shaped metaphors (Figure 1b) and thereby enabling comparability of projects through visualization even if the analyzed context differs substantially.

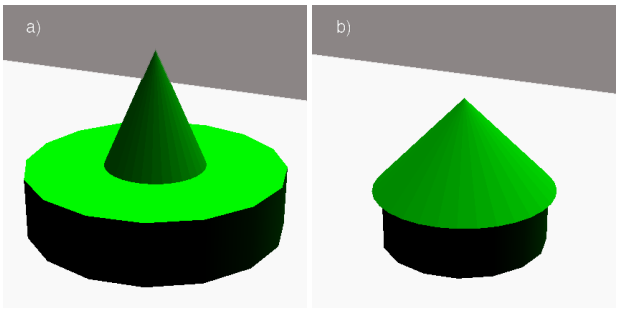

Figure 1. House Metaphor showing a) a miss-shaped b) a well-shaped glyph.

The remainder of this paper is organized as follows. In Section 2 we describe the concepts of the CocoViz visualization, the key visualization and the navigation concepts used to map software metrics to cognitive metaphors. In Section 3 we validate our approach with a case study of the Mozilla project. We discuss related work in Section 4 and finalize with our conclusions and future work in Section 5.

2 CocoViz Visualization

CocoViz maps particular metrics to graphical elements in 2D and 3D following cognitive metaphors. For that it uses a software metrics configurator (to be described later in this section) to deal with appropriate as well as effective metrics combinations and layouts.

Such metrics mapping has already been successfully applied in the RelVis approach [18] and in Polymetric Views by Lanza *et al.* [13]. Based on these work, the CocoViz approach further investigates the usefulness of the third dimension and other possible improvements with respect to the comprehension of a visualized software project.

In the following we consecutively discuss several key visualization and navigation concepts used in our approach. The concepts are explained in the order they build a visualization in CocoViz: 1) Metrics Clusters; 2) metrics configuration; 3) Glyphs; 4) Layouts; 5) Use of 2D and 3D Dimensions; 6) Releases and evolution across releases and 7) Navigation, Tagging of entities (elements) and visualization states.

2.1 Metrics Clusters

Metrics Clusters represent a set of specific metrics which together enable potential explanations for circumstances in the software system under investigaion. Pinzger *et al.* [17] used Metric Clusters as a solution to build condensed views on source code and multiple release history data [5]. Lanza *et al.* [13] used similar concepts in their polimetric views. According to the results in [17], a combination of meaningfully clustered metrics can potentiate the comprehensibility of a software visualization. For example, combining two structural metrics such as *number of functions* and *lines of code* and two complexity metrics such as *Cyclomatic Complexity* [16] and *Halstead Programm Difficulty* [8] accentuate complex software components that exhibit a variety of functionality. Such components are difficult to maintain and evolve and are candidates for design anomalies such as God Classes [6].

Another metrics cluster combines relationship metrics that measure fan-in and fan-out. The resulting view points out components that are more important based on the services they provide. Components providing many services are often more vulnerable to maintenance and often demand for special care.

Metric Clusters are provided as preset mappings for our metric configurator. This offers an easy way to apply a cluster to different visualizations and at the same time allows enhancing or adjusting a cluster with other metrics to satisfy the proper needs.

2.2 Metrics Configuration SV Mixer

The Software Visualization Mixer (SV Mixer) adopts the concept of an audio mixer and equalizer for software visualization. All audio mixers process the level, tone, and dynamics of audio signals with equalizers, filters, limiters, and faders before sending the result to an amplifier. Our software visualization mixer processes the particular software metrics with filters, normalizers, and transformers before composing a visualization. The idea is to quickly adjust the metrics to fulfill our interests while perceiving the view. Similar to the audio mixer channels, every visualization has a specified set of visual representations. To stress the comparability of visualizations in the course of a project the SV Mixer configurations can be saved. Allowing default configuration for a software project to be defined at the beginning and used throughout the whole project. For general purposes preconfigurated presets can be used. We currently investigating on preconfigurated presets for know use cases like the ones described by Marinescu [15].

2.2.1 Filtering Metrics

A visual representation is the visual object (*e.g.,* a rooftop) that represents the metric value in a visualization. A metric can be mapped to a visual representation. To reduce the selected elements to an adequate amount a simple filtering method allows one to specify an interval of interest. An interval of interest is specified for every metric mapping and represents the maximum and minimum value to take into consideration. As in an audio mixer, every channel slider limits the interval of interest for a mapped metric with its minimum and maximum value, offering an easy and fast altering possibility of the data selection. In our SV Mixer, every minimum and maximum slider by default defines eight intervals. In the default configuration the intervals are calculated based on a linear progression from the smallest available value of the mapped metric up to the highest. The slider allows one to quickly reduce the interval of interest by *e.g.,* the highest or lowest 33% of elements.

To calculate the filter threshold value ($threshold$), we use the minimal ($minMV$) and the maximal ($maxMV$) metric values as well as the current ($curSV$) and its maximal ($maxSV$) slider value. The threshold value for every mapped metric is then calculated via linear regression (but not limited to).

$$threshold = minMV + \frac{curSV * (maxMV - minMV)}{maxSV}$$

2.2.2 Normalizing and Scaling Metrics

Beside filtering the selection of data, we further implemented a concept to normalize the mapped metrics to eachother. The idea is similar to an audio mixers pitcher: With the pitcher different audio sources are stretched or condensed to bring each-other in tune. Similarly we want to tune the visual representation values of the mapped metrics to meet a specific project's needs. With the concept of normalization the mapped metric for specific software projects are customized to represent a well-shaped piece of the software as a well-shaped glyph. Furthermore this concept *enables comparability of projects through visualization*

even if the analyzed context differs substantially. As for the filters a mapped metrics normalization factor is adjusted with its slider. In the default configuration a linear function is used. The default normalization value ($scalV$) for each metric is calculated with the minimal ($minMV$) and the maximal value ($maxMV$) as well as the current scalling ($curSV$) slider value of the mapped metric:

$$scalV = minMV + \frac{(maxMV - minMV)}{curSV}$$

Currently, we are investigating if non-linear regression fit as well for normalization.

2.3 Metaphors and Glyphs

Glyphs are visual representations of software metrics. They are generated out of a group of visual representations together with the corresponding metrics mapping. The mapped metrics values then specify the glyphs representation. In the following we introduce glyphs representing a metaphor, that attempt to visualize a software in a comprehensible way, accomplishing a faster cognition of the relevant aspects compared to glyphs without a metaphor (*e.g.,* Starglyphs [4]). A viewer can so quickly distinguish a well shaped glyph from a miss-shaped one. With accurately normalized metrics the resulting visualization intuitively provides orientation about good-designed aspect in the software and distressed ones. In the following we explain three metaphors in further detail. Example visualizations are seen in Figure 3, where the visual representations are mapped to two structural metrics and two complexity metrics.

2.3.1 House Metaphor

The idea of this metaphor is to represent software entities such as classes as houses. A well-designed class then looks like a well-shaped house, whereas a problematic class results in a miss-shaped house. To build such a house metaphor four parameters together with their metrics mapping are used: Two metrics are mapped to the width and height of the roof, whereas the other two metrics represent the width and height of the body of the house. Figure 2 shows clearly some complex classes, represented by the large cylinders. Most of the them have comparably small cone width (*number of functions*) and medium to large cone heights (*lines of code*). Such glyphs represent software components that condense reasonably-sized complex code on few functions. These components might be considered problematic candidates to maintain and evolve.

We investigated visualizations with the house metaphor with 2 and 3 dimensions. In the 2D visualization the roof is drawn as a triangle with metric values for its width and height and the body of the house as a rectangle with further metric values for its width and height. In the 3D representation a cone with its height and diameter represents the roof and a cylinder with its height and diameter the body (Figure 3a). With adequately normalized metrics, a well-designed class results in a cylinder or rectangle with a slightly bigger cone or triangle on top of it (Figure 1b). Problematic classes results in a variety of miss-shaped glyphs (Figure 1a). Interestingly enough in some cases the miss-shaped glyph looks more like other metaphors such as a conifer or fir tree or a church tower. With adequately normalized metrics those special cases of miss-shaped glyphs represent special categories of classes, which can be easily spotted in a visualization.

2.3.2 Table Metaphor

The table metaphor is based on the idea of having software entities such as classes represented as tables. A well-designed class looks like a well-leveled table, whereas a problematic class results in a non-planar table. To build the table metaphor, four parameters together with their metrics mapping are used; each of the metric is mapped to a table-leg (Figure 2b). The table-legs are represented as four cylinders with a rectangle as tabletop placed on top of them. With adequately normalized metrics a well-designed class is perceived as a well shaped table. Problematic classes can easily be recognized in tabletops bevelled to the right, left, front or back. The table metaphor offers an interesting way to perceive strange software components with a simple metahpor-based glyph showing even or leaning tables.

In the table metaphor (Figure 3b) the same cohesion is not as obviously visible as in the house metaphor. The table metaphor is more complex to understand for the viewer, even though the metrics are simply mapped to the table legs. It is hard to distinguish which metric represents which leg while navigating around in the system. Still, with the table metaphor well-formed and miss-formed tables can be easily spotted, what offers a good and intuitive overview where candidates for design anomalies might be and by tagging such candidates they can be quickly found again in other more perceivable views.

2.3.3 Spear Metaphor

In the spears metaphor a well-designed software entity looks like a well-formed spear. To build this metaphor, three parameters together with their metrics mapping correspond to the spear shaft's width and height and to its spike height (Figure 2c). The spear shaft is represented as a cylinder with two spikes represented as cones on both ends of the shaft. A miss-shaped glyph then looks like a very long or a very wide spear. Beyond that representation additional metrics can be mapped on the spear stripes. On the negative

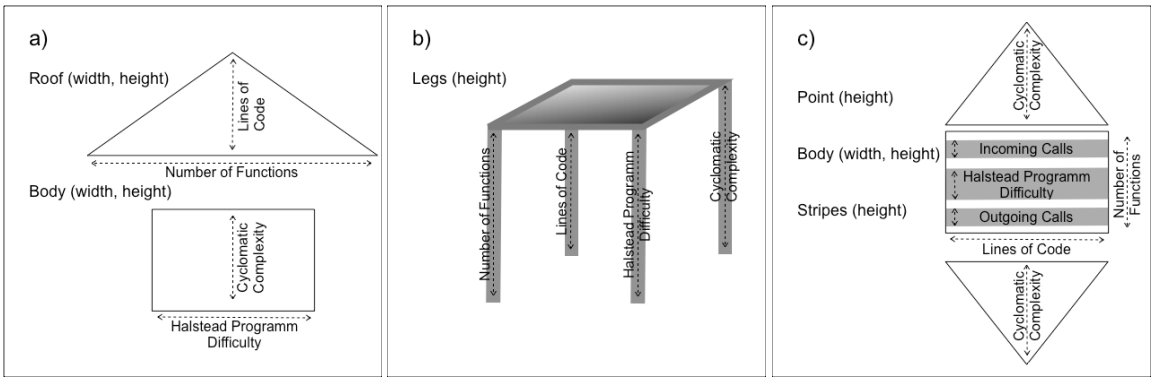

Figure 2. Mapped software metrics to glyphs a) House- b) Table- c) Spear-Metaphor

side, similar to the table metaphor, it does not have any further glyphs, which can additionally categorize the problematic classes. In (Figure 3c) the height of a spear is mapped to the *number of functions*, the width to the *lines of code* and the spear point represents the *Cyclomatic Complexity* metric. All the small fat spears that are visible in the view right away therefore represent large classes with few functions. If they further have a notably sized spear top, they are components that might be considered problematic to maintain and evolve. Additionally to the house metaphor, we mapped access and bug metrics to the stripes. If a problematic candidate beside being large and complex also feature many *bug reports* and many *incoming accesses* (orange to red strips) that emphasize the candidate as already problematic and critical components.

2.4 Layouts

Layouts are an essential part in our approach as in the fewest cases a visualization is built out of one glyph only. In the majority of cases we need to visualize thousands of software entities. To prevent the thousands of glyphs from overlapping or hiding relevant aspects and sustain the comprehensibility we need to layout them accordingly. We investigated different layout algorithms. Basic layout algorithms such as arrangements in circles or chessboard accomplish the intent only to a certain level.

To prevent dispensable overlapping more complex layouts are needed. Furthermore layout algorithms can use visual representations (*e.g.,* coordinate-axes) through which glyphs can be layouted according to mapped metrics. A layout can as well be useful for a specific case only. The primary goal thus is to adequately present a specific set of glyphs based on the situation. In addition to that different glyphs clusters can have their own layout so that glyphs from one model can be arranged differently than glyphs belonging to an other. In our project, the partners from the Università della Svizzera Italiana, Richard Wettel and Michele Lanza, experiment on such layout algorithms that allow one a better observation of the glyphs in a visualization of large sets of software entities [20].

2.5 2D and 3D Dimensions

In our approach the third dimension is used for a general navigation, filtering and selection, before flipping to a detailed representation in a two-dimensional view. We do so as a three-dimensional navigation space has the benefit of dipping into the visualization in a horizontal or vertical view. Software structures can be perceived on a voyage in a virtual world. Nevertheless a third dimensional view has the drawback that too much information and overlapping glyphs can menace to spot relevant aspects. As the metric mapping for the third dimension is not always beneficial we switch to a two-dimensional visualizations whenever we can show proportions and relevant aspects in a more cognitive way. With this approach we stay as flexible as possible and combine the best of the two worlds.

2.6 Releases and Evolution over Time

We visualize time in two different ways, dependent from the amount of input data:
1) We presented all the elements in one view, laid out accordingly on a time axes, whenever changes within evolved software components need to be compared.
2) Whenever a greater amount of components is visualized and confusion arises within the components belonging to different releases, we visualize the data set as a set of visual snapshots. We then can switch through those snapshots and get a small animation like presentation showing the key changes between releases.

As changes made to components between releases are harder to perceive in the snapshots and the all in one view in general offers superior comprehensibility and comparison, we prefer the first approach on small data sets.

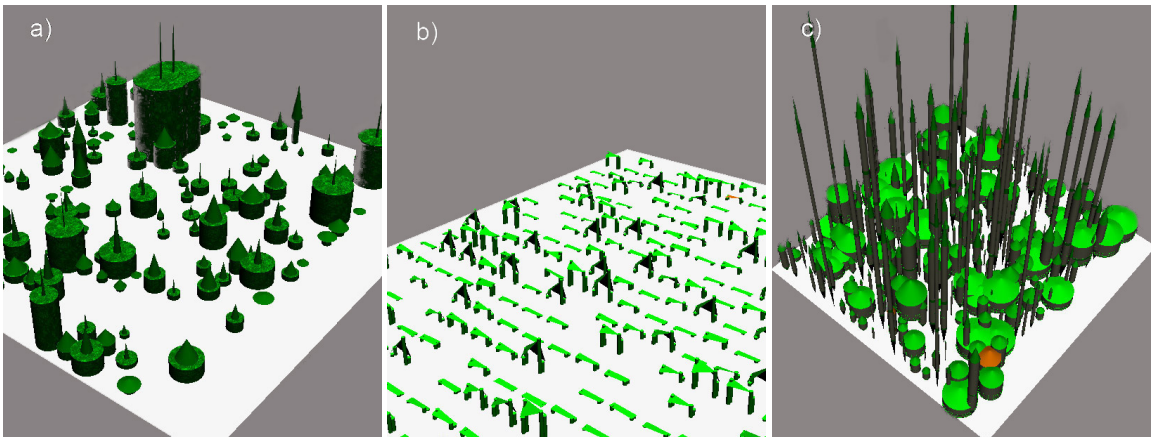

Figure 3. Comparison of not normalized a) House- b) Table- c) Spear-Metaphors

2.7 Tagging Glyphs and Visualization States

In CocoViz we implemented a concept to preserve visualization states or remember interesting glyphs for later analysis. We do so since during the navigation within a visualization, relevant aspects are spotted and need to be remembered before proceeding with the interaction. The remembered aspects can then later be analyzed or shared within the working group. To dynamically interact in a three-dimensional view such a functionality becomes far more important as it allows one to mark the interacting trail and prevents from getting lost within the visualization. Furthermore such a functionality allows to switch to other visualizations and further examine the spotted aspects from another view. Our approach offers two interoperability modes for this issue.

1) We offer one to tag an interesting element while navigating through a view. With this interoperability specific elements can be remembered for later analysis or visualization in another view.

2) Allows one to save the actual view like a snapshot. A snapshot is then used to go back to a certain state or to share spotted aspects within the working group.

3 Case Study

To validate our CocoViz approach we visualized the different metaphors with data sets used in our other works. We use Mozilla as the case study; in particular the same data set as utilized in similar projects by Pinzger *et al.* [18] and Fischer *et al.* [5]. The data set contains the full Mozilla open source project program code with around 1.7 million lines of code. The used evolutionary data set consists of seven Mozilla releases from version 0.92 to version 1.7. The metrics were calculated per release.

In the following we have a closer look at the Mozilla Project, using the spear metaphor. We start our inspection with a size-complexity-mapping (System Hot-Spot-View). This view shows us complex software components that condense a variety of functionality. As stated before such components are difficult to maintain and evolve and are candidates for design anomalies such as God Classes. Figure 4 on the left shows an overview of the version 1.7. At a first glance we note two extreme types of spears. The small long and the fat large spears. The small long spears represent classes having a huge amount of functions but being notably small in size. Like *nsHTMLTableCellElement* (Figure 4a) , which is a part of the complex Document Object Model (DOM)-Module. The class is used to represent HTML content and has 96 functions on 552 lines of code. Classes from that type might as well be more complex (*e.g.,* if they implement an algorithm). They could have many bugs or be critical parts of the application as a central and widely used element. However, they turn out to be simple classes such as interface declarations. To ensure that our classes are less critical, we us the SV-Mixer to temporary filter out the classes that are less complex. We filter out classes that have a history of none to very few critical or non-critical bugs and have few incoming access calls.

The second extreme type of classes, represented as large fat spears, has a notable amount of functions and is large in size. These candidates are considered more problematic to maintain and evolve. To see the potential candidates we filter the classes based on their lines of code and hide the smallest 30%. We further want to know which of these are complex and therefor critical, we do so in applying a filter based on a complexity metric. We can also argue that incoming access calls are more important, as they represent how the classes are used in the system, and therefor filter on an incoming access call metric. With that we reduce the classes to five critical candidate. To verify whether those

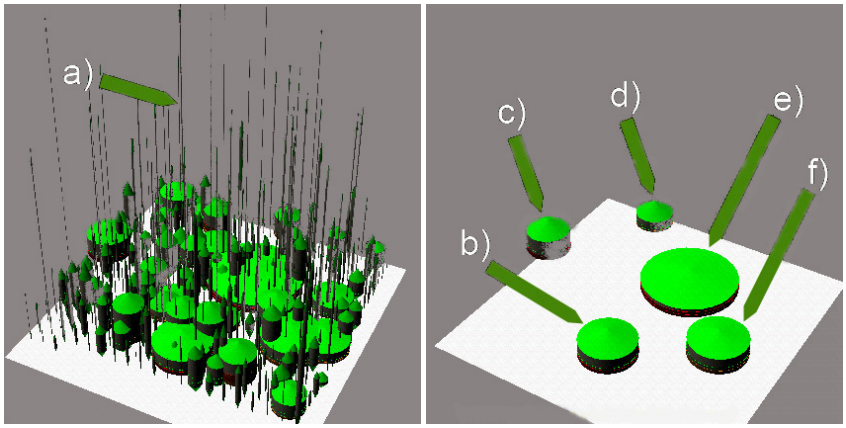

Figure 4. Spear Metaphor showing a system hot-spot-view of Mozilla version 1.7: left shows an overview, from where potential candidates were sequentially filtered out.

candidates really are critical for the Mozilla project, we take a closer look at our metrics and the information provided by the official Mozilla developers site[1].

The *nsGlobalWindow* (Figure 4c) as the previously mentioned *nsHTMLTableCellElement* is part of the complex Document Object Model (DOM)-Module. The class is used to open windows with the represented HTML content and its interaction. It has 230 functions on 6193 lines of code. It accesses 36 functions from other classes and its functions are used by 7 functions from other modules. We further note that it has over 100 non trivial problemreports. *nsGlobalWindow* clearly is a critical component.

nsSelection (Figure 4b) and *nsCSSParser* (Figure 4d) are found within the DOM-Content-Module and deal with the interaction of selections and the parsing CSS files. With 211 functions on 7749 lines of code and 112 functions on 5530 lines of code, they are obviously not small either. Both have over 40 / 20 non trivial problem-reports. *nsSelection* might be less critical as it does not provide basic functionality and is accessed only from one other class.

The *nsPresShell* (Figure 4f) is found in the layout engine HTML-module, which Mozilla uses for rendering tree construction, layouts, paintings, etc. The presentation shell (*nsPresShell*) is used for arena allocations and response to user or script actions like window resizing, document's default font changes or drag & drop operations. It has 226 functions on 8013 lines of code. It uses 242 functions from other classes and its functions are accessed by 7 functions from other modules. In addition to that it has over 130 non trivial problem-reports. Clearly *nsPresShell* is a central and critical component too.

Last but not least *nsCSSFrameConstructor* (Figure 4e) within the same layout engine HTML-Module handles CSS Frames. It has 210 functions on 13494 lines of code. It uses 813 functions from other classes and is accessed by 10 functions from other modules. With over 200 non trivial problem-reports it is obviously at least as critical as the previously mentioned classes.

Within our quick tour through the case study we still have not focused on many details yet. Further options are 1) tagging the elements of our interest and further analyze them in other visualizations; 2) inspect whether there were surprisingly many changes to particular classes over the last revisions; 3) compare the tagged elements by applying other metric-cluster. Nevertheless we showed that with few simple steps we can focus on the essential parts of a system and how CocoViz with its simplicity lets us interact rapidly with profound, complex source-code data.

Results Our first validation with evolution data from the Mozilla case study can be summarized as follows:
1) The SV Mixer allows fast focusing on component qualities of our interest; 2) With the applied metaphors, glyph artifacts can be differentiated rather intuitively; 3) With the house and spear metaphors problematic candidates where quickly perceivable; 4) Tagging candidates allows one to easily interact in different visualizations; 5) Further metrics mapped on the glyphs can help to emphasize critical components

4 Related Work

Software visualization aims to visually represent the complex context of today's software projects. Due to the amount of information that needs to be understood a visualization is crucial. Most of the visualization methods use a graphical representation of data rendered either in a two-dimensional or three-dimensional view. In the past few

[1] http://www.mozilla.org/owners.html

years a variety of approaches dedicated to software visualization and software reengineering has emerged.

Hierarchical visualization approaches tempt to display large hierarchies in a comprehensible form. Johnson and Shneidermann proposed Treemaps [11] to map tree structures on to rectangular regions. The efficient use of space allows to display very large hierarchies with thousands of leaves, while still being comprehensible. However the readability decreases if very large hierarchies are displayed.

In contrast to the concept of Treemaps, Robertson *et al.* [19] suggested Cone Trees. In their work they laid out the hierarchy in a three-dimensional way, where the children of a node are placed evenly spaced along a cone base. Through rotation of the cone base a viewer brings different parts of the tree into focus. But, as stated in [12] Cone Trees with more than 1000 nodes are difficult to manipulate. Therefore, Cone Trees might be considered for medium-sized trees only.

In [2] Dachselt and Ebert recommend an interaction technique for medium-sized trees: The Collapsible Cylindrical Trees (CCT) map the child nodes on a rotating cylinder. This offers a fast and intuitive interaction and allows one to dynamically hide or show further details. The interesting part of this work is that beside most other work in the field of hierarchical views they do not concentrate on how to display large hierarchies in a comprehensible form but concentrate on the interaction with the data itself.

CocoViz distinguishes itself from the hierarchical visualizations among other things in that it uses a 3D view to avoiding space limitations. Appropriate layout algorithms to prevent dispensable overlapping and an advanced dynamic approach that allows intuitive interaction.

Metrics visualization in contrast to hierarchical visualizations, describe a software state or situation. Metrics are not part of a hierarchy, but they describe a specific software entity. The goal of this approach is to show aspects of a software by visualizing the representing metrics.

Eick's Seesoft [3] mapped the lines of code of every software entity to a thin row. The rows are then colored based on a statistics of interest, *e.g.,* most recently, least recently changed, or locations of characters. With that, one can quickly overview the fragmentation of a software and highlight parts of interest. In [14] Marcus *et al.* extend the Seesoft approach by utilizing the third dimension, and by adding different manipulation techniques. They use cylinders where the height, depth, color, and position would represent the metrics.

Polymetric Views of Lanza and Ducasse [13] attempt to help understanding the structure of a software system and detect problems as early as possible in the initial phases of a reverse engineering process. In their concept they display the software entities based on their metric values as a rectangular shape. Whereas the position, the height, the width and the color of one rectangle each represents a metric value of the same software entity. This approach offers a quick overview of the softwares subdivision. Compared to Seesoft the Polymetric Views additionally includes a representation of the relations within the software entities.

Inselberg and and Dimsdale presented a way to visualize multi-dimentional analytic and synthetic geometry [9]. In the parallel coordinates, they arrange the various metric scales vertically one after the other. For every software entity the metric values are marked on the corresponding metric scale. A line connecting all the marks of one entity then represents that software entity.

For several years researchers tried to refine the parallel coordinates to address the limitation of displaying large data sets. Fua *et al.* [7] proposed a multi-resolution view of the data. With this approach it is possible to navigate through a structure by hierarchically clustering a certain level of detail. In [1] Benedix *et al.* explain how the layout of parallel coordinates can be used to visualize categorical data. In their adoption the data points are substituted with a frequency-based representation offering auxiliary efficient work with meta-data. Johansson *et al.* [10] extended the standard parallel coordinates to the third dimension. In their Clustered Multi-Relational Parallel Coordinates (CMRPC) they propose a technique that offers a simultaneous one-to-one relation analysis between a selected dimension and the other dimensions.

In [4] Fanea *et al.* combined parallel coordinates and star glyphs to provide a more efficient analysis compared to the original parallel coordinates. Pinzger *et al.* proposed to use star glyphs to visualize condensed graphical views on source code and relation history data [18]. In their Kiviat diagram, metric values of different releases are reflected like annual rings on a tree-stump. The diagrams can be used to show one metric in multiple modules or multiple metrics in one module. Furthermore relation of modules are characterized with connections between those modules.

CocoViz distinguishes itself from the other metrics visualization approaches through its metaphor glyphs an the resulting improved software comprehension compared to abstract graphical representation used in other approaches. An interactive approach where a viewer analyses the software in walking through the views and tagging elements. And last but not least a dynamic approach that allows to quickly filter non relevant elements out.

5 Conclusions & Future Work

In this paper we discussed how the perception of relevant aspects in evolved software projects can be improved.

Based on previous work in visualization, we filtered key concepts for software visualization, and considered improvements on these key concepts.

We introduced the concept of a software metrics configuration mixer, which brings the benefits of an audio mixer to software visualization. This SV Mixer offers a fast access to the metric configurations and allows one a quick situational accommodation to the visualizations while observing it. Relevant aspects in the software are experienced in swiftly filtering temporary irrelevant data out and tagging relevant aspects for later analysis. In the SV Mixer concepts such as metric clusters are as well incorporated in a straightforward way offering shifts in the visualization right on the filtered aspects of interest. Beyond that, specific metric configurations of a particular project can be predefined and shared within a project team.

We further proposed that software is better understood if its metric based analysis is visualized in cognitive perceptible metaphors. A metaphor maps the metric values of the analyzed software to visual representations which together represent a software entity as a glyph. Good-designed aspects of a software can be distinguished much faster from distressed ones if the used glyph metaphor represents one of the states in an well-shaped object known to the observer. With the house, table, and spear metaphor, we introduced possible implementations of such metaphors.

Future work aims to identify further improvements on the key concepts used in software visualization with evolutionary data. One line of research will focus on augmenting the SV Mixer's capabilities with advanced filter capabilities. Filter configurations will be enhanced with statistical information and the observed data subsets dynamically prepared while observing the visualization. Further investigations will be conducted to produce other cognitive metaphors that can facilitate the perception of the analyzed software in specific use cases. Our approach will be evaluated in comparison to additional software projects. We will further evaluate our approach against other known visualizations to document in which situations we gain advantages over the others.

6 Acknowledgments

We are grateful to M. Fischer, G. Reif, M. Pinzger, and the anonymous reviewers for their valuable input. This work was partially supported by the Hasler Stiftung Switzerland.

References

[1] F. Bendix, R. Kosara, and H. Hauser. Parallel sets: visual analysis of categorical data. *IEEE Symp. on Info. Visualization*, pages 133–140, 2005.

[2] R. Dachselt and J. Ebert. Collapsible cylindrical trees: A fast hierarchical navigation technique. *IEEE Symp. on Info. Visualization*, pages 79–86, 2001.

[3] S. G. Eick, J. L. Steffen, and E. E. Sumner, Jr. Seesoft - a tool for visualizing line oriented software statistics. *IEEE Trans. Softw. Eng.*, 18(11):957–968, 1992.

[4] E. Fanea, S. Carpendale, and T. Isenberg. An interactive 3d integration of parallel coordinates and star glyphs. *IEEE Symp. on Info. Visualization*, pages 149–156, 2005.

[5] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proc. Int'l Conf. on Softw. Maintenance*, pages 23–32, 2003.

[6] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[7] Y.-H. Fua, M. O. Ward, and E. A. Rundensteiner. Hierarchical parallel coordinates for exploration of large datasets. In *Proc. Conf. on Visualization*, pages 43–50, 1999.

[8] M. H. Halstead. *Elements of software science, operating and programming system series*. Elsevier, 7, 1977.

[9] A. Inselberg and B. Dimsdale. Parallel coordinates: a tool for visualizing multi-dimensional geometry. In *Proc. IEEE Conf. on Visualization*, pages 361–378, 1990.

[10] J. Johansson, M. Cooper, and M. Jern. 3-dimensional display for clustered multi-relational parallel coordinates. *Int'l Conf. on Info. Visualization*, pages 188–193, 2005.

[11] B. Johnson and B. Shneiderman. Tree-maps: a space-filling approach to the visualization of hierarchical information structures. In *Proc. IEEE Conf. on Visualization*, pages 284–291, 1991.

[12] J. Lamping, R. Rao, and P. Pirolli. A focus+context technique based on hyperbolic geometry for visualizing large hierarchies. In *Proc. SIGCHI Conf. on Human factors in computing systems*, pages 401–408, 1995.

[13] M. Lanza and S. Ducasse. Polymetric views — a lightweight visual approach to reverse engineering. *IEEE Trans. on Softw. Eng.*, 29(9):782–795, 2003.

[14] A. Marcus, L. Feng, and J. I. Maletic. 3d representations for software visualization. In *Proc. ACM Symp. on Softw. Visualization*, pages 27–36, 2003.

[15] R. Marinescu. Detection strategies: metrics-based rules for detecting design flaws. In *Proc. IEEE Int'l Conf. on Softw. Maintenance*, pages 350–359, 2004.

[16] T. J. McCabe. A complexity measure. *IEEE Trans. on Softw. Eng.*, 2(4), 1976.

[17] M. Pinzger. *ArchView - Analyzing Evolutionary Aspects of Complex Software Systems*. Vienna University of Technology, 2005.

[18] M. Pinzger, H. Gall, M. Fischer, and M. Lanza. Visualizing multiple evolution metrics. In *Proc. ACM Symp. on Softw. Visualization*, pages 67–75, 2005.

[19] G. G. Robertson, J. D. Mackinlay, and S. K. Card. Cone trees: animated 3d visualizations of hierarchical information. In *Proc. SIGCHI Conf. on Human factors in computing systems*, pages 189–194, 1991.

[20] R. Wettel and M. Lanza. Program comprehension through software habitability. In *Proc. Int'l Conf. on Program Comprehension*, 2007.