Software Visualization with Audio Supported Cognitive Glyphs

Sandro Boccuzzo and Harald C. Gall
Department of Informatics, University of Zurich, Switzerland
{boccuzzo, gall}@ifi.uzh.ch

Abstract

There exist numerous software visualization techniques that aim to facilitate program comprehension. One of the main concerns in every such software visualization is to identify relevant aspects fast and provide information in an effective way. In previous work, we developed a cognitive visualization technique and tool called CocoViz that uses common place metaphors for an intuitive understanding of software structures and evolution. In this paper, we address software comprehension by a combination of visualization and audio. Evolution and structural aspects are annotated with different audio to represent concepts such as design erosion, code smells or evolution metrics. We use audio concepts such as loudness, sharpness, tone pitch, roughness or oscillation and map those to properties of classes and packages. As such we provide an audio annotation of software entities along their version history for software analysis and software browsing. Our first results with the prototype and a small user study show that with this combination of visual and aural means we can facilitate program comprehension and provide additional information that usually is not provided by current visualization approaches.

1 Introduction

With the increasing complexity of software systems, program comprehension is a major concern in maintenance and evolution. The amount of data, the relationships between the entities, and missing or out of date documentation make it almost impossible for engineers to maintain an accurate understanding of an evolving system without effective tool support. A variety of project stakeholders are interested in different aspects of a system. A project manager for example might not be interested in the entire system, but only in a reflection of the state of a project. Other stakeholders such as auditors or customers might want to have deeper insights into the project, while not even being allowed to access the source code. There is an opportunity in providing them with visualizations that support their work and offer a quick and comprehensive status on a software project. Such a software visualization needs to aggregate all the gathered information about a project in an effective visual representation.

With the CocoViz project we aim to enhance existing maintenance and evolution analysis methods to present a software system in an intuitively understandable visualization. The benefit of such a cognitive visualization lies in representing the context of interest with perceivable glyphs, abstracting from the real complexity of a system. The different stakeholders can analyze a system within their particular context.

Perceivable glyphs could be any object known from our day to day lives. In our recent work we focus on a house metaphor in which a well-shaped house represents a well-designed software entity and a miss-shaped house shows evolutionary decay. To further improve the perception of our software visualization we looked at multimedia content we use in our daily lives. The perception of multimedia content nowadays is often supported with audio: From a movie, in which the dramaturgy is enhanced with music, to games that produce a virtual reality, or to the interaction with a computer operating system. All these support their visual content with appropriate audio. Therefore, we started to investigate in what way software visualization can benefit from the addition of audio.

In this paper, we describe our approach of annotating visual cognitive glyphs with diverse audio concepts, and present our first results on how software comprehension tasks are improved by using software visualization enriched with audio.

The main contribution is an extended CocoViz [7] visualization approach, where we implemented the described audio concepts to show the benefits of an audio-supported software visualization. The tool architecture was extended to include information gathered with various other software analysis techniques into a single cognitive software visualization.

The remainder of this paper is organized as follows. Section 2 covers the key visualization and navigational concepts, used in cognitive software visualization to map software metrics to cognitive glyphs, as far as needed to under-
stand their application in the extended audio context. In Section 3 we describe our approach to support software visualization with audio, with regard to specific tasks of program comprehension and navigation. In Section 4 we discuss concepts for interacting with an audio supported software visualization. Section 5 discusses how we implemented audio in our cognitive software visualization approach. In Section 6 we present example scenarios based on a commercial web application, and in Section 7 we describe the results of a small user study. We address related work in Section 8 and summarize with our conclusions and future work in Section 9.

2 Cognitive Software Visualization (CSV)

In cognitive software visualization structural and evolutionary metrics are mapped to graphical elements in 2D and 3D as cognitive glyphs. The concept of mapping metrics has already been described in [30] and was introduced as Polymetric Views by Lanza et al. [19]. With the CocoViz approach we investigate the usefulness of the third dimension and other improvements with respect to the comprehension of a visualized software project.

In the following we discuss the key concepts in a cognitive software visualization that are later used in the audio context. 1) Metrics Clusters; 2) metrics configuration; and 3) Glyphs.

2.1 Metric Clusters

We define a Metrics Clusters to be a set of specific metrics that in combination enable analysis of particular software entities in terms of their structure (i.e., size or complexity) and evolution (i.e., change coupling or bug density). A similar concept is used in [29] where Pinzger et al. offer a solution to build characteristic views on source code evolution. According to [29], a combination of meaningfully clustered metrics can facilitate the comprehensibility of a software visualization. In a Hot-Spot-View, for example, the metrics cluster consisting of number of functions, lines of code, Cyclomatic Complexity [23] and Halstead Program Difficulty [15] accentuates complex software components that exhibit a variety of functionality.

In CocoViz we implemented a list of Metric Clusters as preset mappings for our metric configurator. With that we are capable to define clusters independently from the different visualizations. The power of Metric Clusters becomes clear if we use our capabilities to import analysis data. We can easily combine results from completely different analysis methods to new metric clusters and visualize the new context according to our needs. Currently CocoViz supports importing data from the Eclipse\(^1\) metrics plugin\(^2\) and Together’s\(^3\) metric analysis functionality. However the architecture is extensible for other data sources.

In the audio context this concept becomes handy as it allows one to combine metrics suitable for a specific audio algorithm to metric clusters that can easily be applied afterwards.

2.2 Metrics Configuration and SV-Mixer

The next key concept we address in this section is the Software Visualization Mixer (SV-Mixer). The SV-Mixer adopts the concept of an audio mixer for software visualization. An audio mixer processes audio signals before sending the result to an amplifier. The same way the SV-Mixer maps the particular software metrics to the visual representations of a cognitive glyph. The metric values are filtered, normalized or transformed according to the SV-Mixer configuration before composing a visualization. Similar to the audio mixer channels, every visualization has a specified set of visual representations. The idea is to quickly adjust the visual mappings according to our focus while looking at the view.

In our extended audio context, the SV-Mixer in addition to the set of visual representations has a set of audio representations. Depending on the specific amount of metrics used from an applied audio algorithm, the corresponding amount of audio representation channels becomes available in the SV-mixer.

2.3 Cognitive Glyphs

Glyphs are represented as a set of visual representations mapped to software metrics (e.g., the hight of a house roof is mapped to lines of code). The mapped metric values of a specific entity specify the glyphs representation. Beyond that, cognitive glyphs visualize software in a comprehensible way, leading to a faster comprehension of the relevant aspects compared to glyphs that are not based on a metaphor (e.g., Starglyphs [13]).

To build a house glyph, for example, four parameters together with their metric mappings are used. Two metric mappings represent the width and height of the roof, where as other metrics are mapped to the width and height of the body of the house. In the context of a Hot-Spot-View Metric Cluster the example in Figure 1a) represents a complex class, visualized with a large house body, and a comparable small house roof width (number of functions) and a medium to large roof height (lines of code) the glyph would represent a software component that condense reasonably-sized complex code on few functions like a class implementing an complex algorithm. These components might be considered problematic candidates to maintain and evolve.

\(^1\)http://www.eclipse.org/  
\(^2\)http://metrics.sourceforge.net/  
\(^3\)http://www.borland.com/us/products/together/index.html
In our audio extension, every cognitive glyph representing a software entity has its audio context. Based on the comprehension or analysis task, the audio context offers further details on the particular software entity.

3 Why Audio in Software Visualization

One of the main concerns in a software visualization remains to find relevant aspects in a complex system as fast as possible. Within a Cognitive Software Visualization (CSV) as explained in [7] the glyphs representing the various entities and their aspects are distinguishable and perceivable. The SV-Mixer allows one to customize the view and interact with it. Still there are cases in CSV as well as general in software visualizations, where after filtering out irrelevant entities, one finds himself with hundreds of potential entities. In many layouts these entities even overlap each other.

To overcome these shortcomings and further improve our CSV approach, we were looking for components that are fundamental in program understanding and navigation. We want to mention some of the work we found important for our purpose. According to Pennington’s [27, 28] bottom-up theory of program comprehension e.g., a programmer focuses first on the basic structural entities. A fundamental component therefore is an adequate highlighting of basic text structure units. Mosemann and Wiedenbeck’s results presented in [24] stated that reading a program by following the control flow offers a high performance way of navigation even for novices.

In [25] Pacione proposed ways to increase the utility of visualization for software comprehension. He classified visualization to five levels of abstraction for software comprehension. Pacione suggested that software comprehension can be facilitated in adequately using multiple of those levels of abstraction combined with multiple facets and the integration of static and dynamic information. Pacione et al. did a case study in a realistic software comprehension scenario [26]. According to them visualizing an object- or class-level representation of the system and providing an architectural-level view were optimal in terms of answering most of the scenario questions. With our cognitive software visualization approach we place ourself already in the bottom-up theory of program comprehension (e.g., a programer focuses first on the basic structural entities. A fundamental component therefore is an adequate highlighting of basic text structure units. Mosemann and Wiedenbeck’s results presented in [24] stated that reading a program by following the control flow offers a high performance way of navigation even for novices.

In [25] Pacione proposed ways to increase the utility of visualization for software comprehension. He classified visualization to five levels of abstraction for software comprehension. Pacione suggested that software comprehension can be facilitated in adequately using multiple of those levels of abstraction combined with multiple facets and the integration of static and dynamic information. Pacione et al. did a case study in a realistic software comprehension scenario [26]. According to them visualizing an object- or class-level representation of the system and providing an architectural-level view were optimal in terms of answering most of the scenario questions. With our cognitive software visualization approach we place ourself already in the bottom-up theory of program comprehension (e.g., a programer focuses first on the basic structural entities. A fundamental component therefore is an adequate highlighting of basic text structure units. Mosemann and Wiedenbeck’s results presented in [24] stated that reading a program by following the control flow offers a high performance way of navigation even for novices.

In our audio extension, every cognitive glyph representing a software entity has its audio context. Based on the comprehension or analysis task, the audio context offers further details on the particular software entity.

4 Audio Concepts

Audio has been used for software analysis in previous work e.g., in [10] to monitor control flow or in [1] to provide programmers with debugging feedback. In the context of software visualization we use audio for tasks, in which we can effectively support interaction and navigation. In this section we discuss our extended visualization with audio support.

4.1 Audio in Data Analysis

Even after filtering out irrelevant data the remaining visualized data often contains a variety of entities. It becomes important that one can get extended information about a particular entity. To do this in a traditional visualization one would represent the entity in a different view, loosing the focus from the original visualization. In supporting the visualization with audio we can provide the user with additional information, while preserving the current visualization state.

4.1.1 Detect threshold exceeding with audio

A simple way to provide a user with extended information is to check a mapped metric for exceeding thresholds. Whenever an entity exhibits a metric value beyond a specified value, a sound (e.g., in the simplest case an acoustic signal) would provide the information to the user. This approach is convenient in cases where we are looking for entities that are outliers. For instance we visualized a system with a Hot-Spot-View (by applying the size-complexity-metrics cluster preset) and were interested in entities for which many critical bugs were reported. With threshold exceeding audio annotation we can map Nr of critical Bugs to the entities and get notified which entities in the Hot-Spot-View are critical.

With a slightly modified version we are able to track not only threshold exceeding but also intervals. That allows us to simply classify the entities based on a mapped metric. The advantage is that we can give an audio feedback to which subgroup an entity would belong in an extended or
4.1.2 Detect code smells with audio

Another way to provide a user with extended information is to check for code smells in entities of interest. Different approaches exist to find code smells or anti-patterns in source code. In our work, we use the approach developed by Lanza and Marinescu in [20]. In a traditional approach the entities with detected code smells would be visualized separately. For example, we would color the detected entities according to a color concept (Figure 2). The bottleneck in coloring the entities based on their detected code smells is that we cannot apply another color concept to the data set, and even worse that whenever an entity includes two or more different code smells, we would need to color the entities with a default color. Unfortunately, a default color leaves us with just little, and imprecise new information. We still would need to further dig into the entity to see what code smells were detected.

In an audio-supported visualization, an entity can give us a more precise audio feedback. The feedback could be, for example, spoken or non-spoken, depending on the need. For instance clicking on an entity would trigger the audio feedback. The entity would then speak its extended information, like ‘the current entity is a potential god class and a potential shot gun surgery class’. Of course, spoken text is only a rather simple example of audio support. Harmonies, disharmonies, twang or shrillness are other possibilities to enrich an entity’s audio-supported visualization. Compared to the visual only approach we can get a very informative summary of the entity’s extended context including cases where entities incorporate two or more code smells.

4.1.3 Entity description with synthesized audio

An even more sophisticated approach to use audio in data analysis is to synthesize an audio feedback based on the related entities values. The important part in such an approach is that the audio feedback needs to maintain its distinction from other similar entities. A lot of work has been done in the field of psycho-acoustics to address the issue of distinguishability, especially in the context of audio compression. Particularly worth mentioning in our context are the so called Zwicker parameters [36]. According to Zwicker et al. the parameters loudness, sharpness, tone pitch, roughness and oscillation do preserve the distinguishability of synthesized audio.

With synthesized audio we can generate a highly complex feedback that still remains compact. In an implementation of a synthesized audio algorithm we map metric clusters to the various Zwicker parameters.

**Example 1: A large and long lived entity with few critical bugs and low coupling.** As an example, we map the value of the entities for FanIn to loudness, the Number of critical Bugs to roughness, the Historical growth rate of the entity to tone pitch and the Lines of Code to the length of the tone. With that a large entity with few critical bugs, that is not used often by other classes and has been in the system for quite some time, produces an audio feedback of a long, clear, but rather low tone.

**Example 2: A large but young entity with many critical bugs and high coupling.** Such an entity that is widely used by other classes but was introduced only in recent releases and incorporates several critical bugs would result in a long, rough, shrill and loud tone.

The advantage of our audio to metric cluster mapping is that we can provide the user with quite a bit of extended information including historical information as well as current health and importance of an entity with effective acoustical feedback.
Table 1 shows two possible mappings of metric clusters to the Zwicker parameters that we found useful as described in the examples above.

<table>
<thead>
<tr>
<th>Zwicker Parameter</th>
<th>Structural Metric Cluster</th>
<th>Evolution Metric Cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loudness</td>
<td>Complexity</td>
<td>Fan in</td>
</tr>
<tr>
<td>Tone pitch</td>
<td>Growth rate</td>
<td>Growth rate</td>
</tr>
<tr>
<td>Roughness</td>
<td>-</td>
<td># critical Bugs</td>
</tr>
<tr>
<td>Oscillation</td>
<td>-</td>
<td>Change rate</td>
</tr>
<tr>
<td>Tone length</td>
<td>Lines of Code</td>
<td>Lines of Code</td>
</tr>
</tbody>
</table>

Table 1. Example of two metric clusters mapped to synthesized audio concepts

4.2 Audio in Historical Analysis

Besides using historical information to synthesize an audio feedback, audio offers support for other historical analysis tasks. Whenever we change the focus of a data set from one version to another in traditional visualizations we encounter a variety of changes happening at the same time. Depending on the layout applied to a view this results in objects changing position, objects disappearing or shrinking substantially in size. It can be hard to keep track of a particular set of interesting entities during such changes. Traditional visualizations animate the entities to represent their change in position and size from one version to the next. Still, with a variety of changes happening at the same time, it remains hard to perceive whether an entity of interest changed substantially from one version to another.

With the use of audio we can support this visualization in simply notifying whenever an entity of interest changes more than e.g., 10% from its previous version. The audio feedback could again be spoken or non-spoken, depending on our preference. For example, the evolutionary change of entities from one version to another could give a simple spoken annotation such as ‘from version 1.0 to 2.0 the tagged entities ListProducer and PersistentModel changed by more than 10 percent’.

4.3 Audio in Trace Analysis

Within our EvoSpaces project\(^4\) we found visual tracing to be an informative way to analyze use cases [11]. The bottleneck of visual tracing is that with the endless amount of actions one can get lost in a wonderful animation. Especially in cases when we aim to understand how often and from where specific entities are called.

To overcome this we suggest to use audio in the visual tracing context similar to the one used by Baecker et al. in [1] to provide programmers with debugging feedback. Before visualizing the trace we tag the entities of interest. With that we can keep track of and get notified on the interactions that the particular set of entities is engaged in, while the visualization shows us the system interaction during our use case as a whole.

5 Audio Supported Cognitive Glyphs

In our CocoViz approach we implemented audio support on the level of supporting navigation to achieve getting fast end effective familiarity with program entities and collecting information about a program as referred in [24]. This is done by binding audio feedback to the results of algorithms that gather extended information on specific program entities in an actual visualization.

\(^4\)http://www.inf.unisi.ch/projects/evospaces/
We found such audio feedback particularly useful in cases where even after adequately filtering visual entities in the SV-Mixer we still ended up with hundreds of potential entities relevant in the context of interest or whenever we need to fine-tune the visualized entities. In such cases an adequate context-oriented audio feedback can improve interaction with a cognitive software visualization substantially, without the need of changing the visualization and loosing subsequently focus.

Beyond that we can use audio for finding and highlighting basic text structure units as Pennington suggested in his bottom-up theory of program comprehension [27, 28], and support a programmer in finding and understanding basic structural entities in a large program. With audio feedback we are also able to extend our level of abstraction as defined in [25], to include information from other abstraction levels such as object population, memory usage, load distribution and deployment informations from a microscopic level or informations on business behaviour and use cases from a macroscopic level.

With audio feedback we found a way to improve interaction in cognitive software visualizations, extend the visualization approach to adequately address further software comprehension tasks, while still preserving all the advantages of a non audio-supported cognitive software visualization.

6 Example Scenarios

In this section we present example scenarios with an analysis of a commercial web application. We show situations where audio feedback is particularly useful and compare it to the effort that would be needed to achieve the same result with a non audio supported visualization. The used evolutionary data set consists of the basic framework of a commercial web application used in healthcare. We analyzed six releases over the period of 3 years. The metrics were calculated per release. The framework has more than 950 classes and approximately 90,000 lines of code. In the following, we analyze the framework from a program comprehension and a software analysis point of view.

6.1 Program comprehension task

In an example program comprehension scenario a quality-assurance team could ask for critical classes of the system that where changed since the last release and need to undergo extensive testing. A set of such critical classes that should be included in the extensive testing can be found by looking at potential big and complex classes that are widely used by other classes and have been changed during the last release.

For this we select all the class entities of the application and visualize them using our house glyphs. The visual representations of the house glyphs are mapped to a Hot-Spot-View (size-complexity-metrics) as described in Section 2.1. With that we show complex software components that condense a variety of functionality. We arrange the entities on the visualization axes using their lines of code and weighted methods per class values. The colors in Figure 3a) represent fanOut, a metric showing how extensively the classes use other classes. We mapped and configured the different colors in the SV-Mixer to cluster the entities in a sensible way. Classes using methods from less than 4 other classes are colored in blue. Classes that use up to 8 other classes are colored in orange and classes with more than 8 are shown in red.

Figure 3a) shows the smaller classes of our visualization setup. In the upper right corner of the visualization we see houses that are obviously complex (body size) and large (roof size). In a context of finding the crucial components of our system, we need to take those into consideration for an in-depth analysis. However from a crucial components perspective, we need to pay attention on some of the smaller classes, too. Because they might only be small based on their inheritance.

To gain familiarity with the small but relevant classes, with current visualization approaches we end up having to use a new view. In our approach, for example, we do this by either changing the layout mapping to rearrange the entities based on their inheritance, with the disadvantage of losing orientation and finding all the entities on a new position, or as shown in Figure 3b), changing the mapping of the colors and at least preserve a comparability between the two views. In any case we lose focus on our main context. Furthermore we end up having a variety of less important extra views making the comprehension of a project as a whole more difficult.

With audio feedback, a context like the small crucial classes could be achieved without losing focus on the primary view. For instance we can simply map the inheritance level of the entities to give us an audio feedback based on a threshold exceeding audio algorithm. We then hover with the mouse over the entities of our interest. Whenever there is an entity that we need to take into consideration we get an audio feedback and can select that entity right away.

Finally we get information on which of this classes were changed during the last release and need extensive testing with a historical audio algorithm. With that, classes notify us their change from the last release with two musical notes playing a interval representing the amount of the change.

To emphasize the usability of audio-supported software visualization for program comprehension, we extend our current example to another common use case. Let's consider that we are trying to find a bug that was introduced during the latest releases. We change focus from our entities of interest to the ones changed during the last releases. In a tra-
ditional visualization approach we change the visualization again by mapping the color to the entities changed during the last releases. Even though we do not know whether any of our entities under investigation has been changed during the latest releases, we change focus in our view to gain little or no information.

With audio support we can run our entities through an audio algorithm that notifies us whether the entities were changed during the latest release. With that we again hover over the entities under investigation. We then simply select the ones changed during the latest release that are potentially involved with the bug for an in-depth analysis without losing focus.

6.2 Software analysis task

A common software analysis scenario is to detect code smells. In a traditional CocoViz approach we analyze the entities color by a god classes detection algorithm. As shown in Figure 2 the brighter the color the higher the god class potential of a class.

If we are further interested in which of the potential god classes are affected by another code smell, we can map the color of the entities to another code smell algorithm. Without losing focus we can run the entities through a code smell audio algorithm and hover over the entities under investigation. Immediately, we then can hear with what other code smells the classes are potentially affected.

7 User Study

In this section we present the first results of a user study we have done with 10 individuals not involved within this project. For the user study we used an extended version of our CocoViz-Implementation [6], which implements all the previously mentioned audio concepts.

Characteristic of Participants  The subjects inquired in the experiment where categorized based on their musical education and level of expertise in program comprehension. For musical education, we distinguished individuals that were able to read music sheets, play a music instrument or sing, from the ones that did not have musical experience and further divided them into candidates with and without software engineering knowledge. We did so to check preliminary concerns on whether our audio approach would be harder to understand for individuals with no musical education. Beside that we wanted to know how much training a new user would need to get familiar with the audio approach and to distinguish the entity properties through audio. We took non software engineers into consideration because one reason of the cognitive glyph abstraction, is that important individuals like CFO in smaller companies without an own information system department, understand the situation even though they often are not familiar with source code.

Design  In this user study we asked the participants to address the situation, in which a quality-assurance team wants
to know which critical classes of the system where changed since the last release and need to undergo extensive testing. Like described in Section 6 we want them to find such critical classes that where changed. During the user study all participants used the same data-set of a commercial web application. In the end we asked the Individuals questions like whether the audio supported visualization was useful for them or what benefit they encountered, compared to a visual only approach.

**Preliminary results** of our user study showed that in general the usefulness of the idea was clearly received. In our study all the participants, no matter on their musical education, did not encounter any problems in instantaneous understanding the audio feedback. Our concerns on whether training would be needed was pretty much disproved. For non engineers with the cognitive glyph metaphor, only little explanation of the general cognitive glyph representation was needed, to understand the task, not mentioning the source code at all.

No general preference for speech versus non-speech audio feedback was found. Depending on the task one was preferred over the other and vice versa.

Concerns arise in audio algorithms using the Zwicker parameters[36]. Tone length and loudness were harder to perceive by the individuals in a noisy environment or without headphones. Especially loudness showed itself as being hard to map linear differences correctly, as it seems that every individual does percept loudness different and not linear.

Some individual stated that for certain tasks a popup would be as useful as audio feedback. Still the same one found the idea to support the visualization with an extra dimension that prevents further overloading the screen very promising and even suggested future work to areas that popup can not keep up.

### 8 Related Work

The goal of Software visualization is to represent the complex context of today’s software projects. To visualize software is essential, due to the abstract nature of the context and the amount of information that needs to be understood. Most visualization methods use a graphical representation of data rendered either in a two-dimensional or three-dimensional view. In the past few years a variety of approaches dedicated to software visualization and software reengineering emerged.

**Hierarchical visualization** approaches aim to display large hierarchies in a comprehensible form. With Treemaps [17], Johnson and Shneidermann proposed to map tree structures to rectangular regions. Very large hierarchies with thousands of leaves, can be displayed space efficient, while still being comprehensible. However the readability decreases very large hierarchies and the attraction in the visualization is often centered on relatively unimportant entities that are represented as a large rectangular region.

In contrast with Cone Trees, Robertson et al. [31] suggested to 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 [18] Cone Trees with more than 1000 nodes are difficult to manipulate. Therefore, Cone Trees might be considered for medium-sized trees only.

An other technique for interaction with medium-sized trees Dachselt and Ebert recommend in [9]. 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 CCT do not concentrate on how to display large hierarchies in a comprehensible form but concentrate on the interaction with the data itself.

With our CocoViz approach we aim to avoid the short comings of the mentioned hierarchical visualizations, in combining their key concepts. Among other things we use 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 describe a specific software entity and are not part of a hierarchy. The goal of these approaches is to show aspects of a software by visualizing the representing metrics.

In Seesoft [12] Eick represents the lines of code of every software entity as 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. Marcus et al.’s sv3D[21] extends the Seesoft approach to the third dimension, and adds different manipulation techniques. They use cylinders where the height, depth, color, and position would represent the metrics.

Lanza and Ducasse’s Polymetric Views [19] attempt to detect problems as early as possible in the initial phases of a reverse engineering process and aim to help understanding the structure of a software system. 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. The Polymetric Views in addition to Seesoft include a representation of
the relations within the software entities.

Inselberg and and Dimsdale presented a way to visualize multi-dimensional analytic and synthetic geometry [16]. 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.

In [3] Benedix et al. explain how the layout of parallel coordinates can be used to visualize categorical data. In their approach the data points are substituted with a frequency-based representation offering auxiliary efficient work with meta-data.

In [13] 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 [30]. 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 temporary non relevant elements out.

Audio supported visualization To our best knowledge we found only little work done with audio in software visualization. However there is work done in the context of software analysis and auditory display.

Vickers in [33] gives a good introduction in summarizing various approaches present in the field auditory representation of programs.

Brown and Hershberger in [8] use audio to enhance algorithm animations. In their work they enhance the Zeus algorithm animation systems using a MIDI synthesizer and give an introduction to the use of colour and sound in algorithm animations.

In [10] DiGiano et al. explored with a sound-enhanced programming environment they call LogoMedia. LogoMedia allows programmers to associate non-speech audio with program events while the code is being developed. The interface is designed for specifying visualization events with sound and monitoring variables or control flow.

Baecker et al. in [1] suggest to use audio to provide programmers with debugging and profiling feedback without disturbing the integrity of the graphical interface. According to them audio may be a more salient representation for certain types of program information like repetitious patterns in control flow and nonlinear sequences of variable values.

In [14] Finlayson and Mellish compared speech, non-speech sound and a combination of them. They recommend that a combination where non-speech sound be used as a supplement to speech as it shows slightly better results compared to pure speech or non-speech.

Berman and Gallagher in [5] present techniques to listen to program slices that help software developer in undertaking program comprehension activities.

Recent work with audio in software analysis has been done by Steffik et al. in [32]. In their work they use aural feedback to sonify computer code as an aid to non-sighted programmers.

CocoViz distinguishes itself from the other approaches in that they focus mainly on tracking the value of state variables and control flow during debugging or visualizing algorithms. Meanwhile our focus is more in supporting the interaction within a visualization.

9 Conclusions & Future Work

In this paper we discussed improvements to the perception of relevant aspects in evolving software projects. We proposed an audio extension to our cognitive software visualization approach CocoViz [7], where metric based analysis of entities are visualized in form of cognitive glyphs. With our extended cognitive software visualization approach structural and evolutionary aspects of entities not only are distinguished faster, but we are also capable to combine existing program comprehension techniques with audio-supported visualizations. The audio parameters of Zwicker (loudness, tone pitch, roughness, oscillation and tone length) were mapped to metric clusters. These metrics clusters that are presets in our SV-Mixer of CocoViz can therefore be used to set to sound distinguished entities. As a results, conventional visualization techniques are enriched with particular audio sounds to enable a user a more effective program comprehension scenario without losing context in multiple views.

Based on previous work in visualization, we introduced concepts where our cognitive software visualization substantially benefits from audio support over non audio supported visualizations. Audio feedback is used to support the dynamic interaction with a system’s visualization. Especially in visualizations of large systems with many cognitive perceivable glyphs audio feedback facilitates the understanding of relevant aspects. It does so in allowing to hover over potential entities of interest an get instant feedback or notify a user over relevant changes during an animation or trace view.
The approach is currently evaluated with a large set of software projects and against other known visualization approaches to document in which situations an audio feedback offers substantial advantages over more traditional approaches. We are furthermore working on a user study to get substantial data on the benefits of our audio-supported approach CocoViz.

Future work aims to improve the audio feedbacks with user configuration and will consider more sophisticated audio algorithms and to extend the use cases where traditional visualization tasks benefits from an audio support. With one particular audio algorithm we try to synthesize an ambient sound that would get a feedback based on the code smells or properties presented in the proximate entities. Besides that we are currently experimenting with clustering algorithms and synthesized audio algorithms.

Acknowledgments

We are grateful to Emanuel Giger, Patrick Knab, Michael Würsch and Martin Pinzger for their valuable input. This work was partially supported by the Hasler Stiftung Switzerland within the Project “EvoSpaces II”.

References