Visualization-based Analysis of Quality for Large-scale Software Systems

Guillaume Langelier* Houari Sahraoui Pierre Poulin

DIRO, Université de Montréal
Montréal, QC Canada

ABSTRACT

We propose an approach for complex software analysis based on visualization. Our work is motivated by the fact that in spite of years of research and practice, software development and maintenance are still time and resource consuming, and high-risk activities. The most important reason in our opinion is the complexity of many phenomena related to software, such as its evolution and its reliability. In fact, there is very little theory explaining them. Today, we have a unique opportunity to empirically study these phenomena, thanks to large sets of software data available through open-source programs and open repositories. Automatic analysis techniques, such as statistics and machine learning, are usually limited when studying phenomena with unknown or poorly-understood influence factors. We claim that hybrid techniques that combine automatic analysis with human expertise through visualization are excellent alternatives to them. In this paper, we propose a visualization framework that supports quality analysis of large-scale software systems. We circumvent the problem of size by exploiting perception capabilities of the human visual system.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, product metrics*; D.1.5 [**Programming Techniques**]: Object-oriented Programming; H.5.1 [**Information Interfaces and Presentation**]: Multimedia Information Systems

General Terms

Measurement, Experimentation, Human Factors

Keywords

Software visualization, quality assessment, metrics.

*{langelig | sahraouh | poulin} @iro.umontreal.ca

1. INTRODUCTION

In spite of years of research and practice, software development and maintenance are still time and resource consuming, and high-risk activities [1, 6]. Object-oriented (OO) and related technologies have improved significantly the ease of development and maintenance. Indeed, OO has considerably reduced the gap between user requirements and their implementation in software. In practice however, the cost and the risk remain high compared to the development and the maintenance of other manufactured products. In our opinion, this situation still occurs because many phenomena related to software, such as its evolution and its reliability, are still too complex and less understood. This is echoed by the fact that very limited theory can explain them. It is therefore essential to better understand and model these phenomena in order to increase our control on the development and maintenance activities. In other scientific fields, similar situations are addressed using empirical research based on the classical cycle "observations, laws, validation, theory". As more and more large sets of software data are accessible through open-source programs and open repositories, we have now unique opportunities to empirically study these phenomena [5].

The idea of empirically studying phenomena related to software is not new. Many studies have been conducted, especially by the community of software measurement and quality. However, in spite of several quality estimation/prediction models published in the literature [2, 7], concrete applications in industrial contexts are very rare.

The success to expect from the analysis of software data sets depends on the analysis techniques themselves. Automatic analysis techniques, such as statistics and machine learning, are usually limited when studying phenomena with unknown or poorly-understood influence factors. This is the case of software evolution and reliability for instance. We claim that hybrid techniques that combine automatic analysis with human expertise are excellent alternatives that should improve our understanding of software properties.

Visualization offers powerful tools to develop a better understanding of software quality. It allows automatic preprocessing and presentation of the data in such way that a human expert can identify complex regularities and discontinuities that are usually associated with phenomena occurrences. Proper preprocessing and presentation are therefore crucial to control the size and the significance of the studied data. Indeed, expertise cannot be effective when large-scale systems are observed. As an example, a UML class diagram containing hundreds of classes is very difficult to effectively analyze. In this context, we propose a visualization-based approach for complex system analysis that circumvents the problem of size by exploiting perception capabilities of human visual system.

The paper is organized as follows. Section 2 gives an overview of our visualization framework. This framework is uses two representation levels : class representation and program representation. These two levels are respectively detailed in Section 3 and Section 4. The application to our visualization framework for three types of analysis tasks is described in Section 5. An evaluation of our approach is provided with an experimental study, and discussed in Section 6. Finally in Section 7, we conclude with a discussion on the contributions of our approach, its limitations, and proposed improvements.

2. SOFTWARE VISUALIZATION

Visualizing large-scale software to understand both local and global software properties is a very challenging task. Therefore, the more convivial, efficient, flexible our framework is, the more suitable it should become to analyze, understand, and explain software properties.

Four aspects of our visualization framework are described: class representation (Section 3), program representation (Section 4), navigation (Section 4.2), and data filtering (Section 4.3). As our goal is to analyze large-scale programs, we decided to focus on macro analysis (*i.e.*, class as the basic element). Many previous software visualization systems have concentrated on detailing classes into methods and variables (see for example [12, 14]). They offer fine granularity views of software that is important. These views could be connected to our framework in order to complete the analyses.

A crucial decision when building visualization environments for a category of analysis tasks is to determine which data to visualize and how. Too much data hides structural understanding, and too little neglects potentially important information. An image of cluttered data suffers from occlusions, and a badly distributed/organized data possibly hinders existing links. Finally, the human visual system has studied strengths and weaknesses; good visualization must exploit these natural skills to be successful.

More concretely, because of the intangible character of software [11] and due to the nature of the targeted analysis tasks, we decided to work with abstract information (metrics) extracted from code such as its size, cohesion, coupling, etc. This data is mapped to graphical data such as color, shape, size, orientation, etc., that can be easily perceived by the human visual system.

3. CLASS REPRESENTATION

3.1 Representing the Intangible

Visualizing a program is not an easy task because of the intangibility of code. In fact, code is intended to be understood by humans and computers, and has no concrete reality outside of these purposes. Medical imagery and mechanical simulation are two examples that have real and precise objects to represent them in 3D. Similarly, people comparing data associated with geographical areas in terms of any given variable can directly map their natural coordinates to a support for their visualization. Unfortunately, it is impossible to represent software in its original form because it does not have any.

It is therefore necessary to represent code with some arbitrary figures. We decided to represent a *class* with a geometrical *3D box*. The box has a number of interesting features, its simplicity being an important one. Indeed, a box can be rendered very efficiently, thus allowing us to display a very large number of such entities. This simplicity is also crucial for human perception. Our brain analyzes a scene mainly through quick pattern matching, allowing us to better recognize common forms. The straight, regular, and familiar lines of the box are therefore processed very quickly. This efficiency saves more time for the analysis of other box characteristics such as color, size, and twist.

Our framework currently uses only these three characteristics. Although we experimented with other box characteristics, the results were not as significant. The more graphical attributes we introduced, the more interferences they created on each other, or the more difficult it became to efficiently distinguish differences when a large number of stimuli interacted on the display. Nevertheless, choosing the right amount of information to display is a difficult task, and we are still investigating adding significant dimensions to our graphical representations.

3.2 Software Metrics

Software metrics are powerful tools to link a class with a representation, in our case a 3D box. Firstly, software metrics have quantitative values that can be easily manipulated. Therefore it is possible to apply more powerful statistics on them, as well as straightforward transformations on their values (Section 3.3). Secondly, the metric model is ideal for the primary goal of our research: the understanding and evaluation of software quality.

In order to accurately represent a class, we identified four characteristics that we considered relevant to the study of software quality: coupling, cohesion, inheritance, and size-complexity. Several implicit or explicit software design principles involve these characteristics. For example, it is well accepted amongst the software engineering community that software should demonstrate low coupling and high cohesion. Size and complexity are also relevant to quickly identify important classes or to analyze whether a class is too complex and needs refactoring.

The selected characteristics are captured throughout metrics. Many of them were proposed in the literature. For example, coupling can be measured by CBO (Coupling Between Objects), cohesion by LCOM5 (Lack of COhesion in Method), inheritance by DIT (Depth in Inheritance Tree), and size-complexity by WMC (Weighted Methods per Class) [4]. In the remaining of this paper, we use some of these metrics to illustrate our framework.

3.3 Merging Boxes and Classes

Now that we have decided which characteristics we want to represent and have identified interesting graphical features, we need to determine a correspondence (mapping) between these two sets. A mapping will have a direct impact on the quality of analysis, and depends on the type of analysis tasks we want to perform. While our choice in this paper has been motivated by a number of factors, nothing prevents us to customize an arbitrary mapping by linking any metric with anyone of the graphical characteristics. Hence as an example for our tasks, color, twist, and size are matched respectively to the CBO, LCOM5, and WMC metrics.

The type of metrics has also to match feature properties. In our context, color is a continuous linear scale in hue from blue to red. Classes with low CBO are displayed in blue while those very coupled appear in flashy red; an average CBO results in variants of purple. Twist rotates the 3D box in the plane between 0 and 90 degrees. Classes with low LCOM5 (*i.e.*, very cohesive) are presented as very straight boxes while classes with high LCOM5 lie horizontally. Finally, classes with high WMC are presented as tall boxes and classes with low WMC as small boxes. Size and twist are also continuous and linear. Three examples of class representations are illustrated in Figure 1.

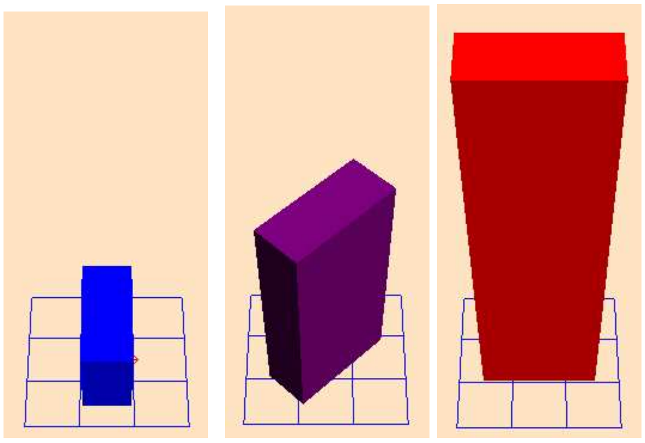

Figure 1: Three class representations: All three metrics (CBO, LCOM5, and WMC) are increasing in value from left to right. Note: most figures in this paper should be viewed in color to better understand their perceptual values. They can be accessed from the website www.iro.umontreal.ca/∼labgelo/publication_material/ase05

This particular mapping is not arbitrary. In addition to provide good perceptual qualities, it has some sort of semantic meaning. Indeed, high coupling is considered bad in software development, and it is generally accepted that the color red means danger. So the presence of red in an area of the visualization can be interpreted as a possible danger represented by that portion of code. Also, twist is well suited for cohesion representation. Again, the association of being straight with coherence and correctness is generally well accepted. A twisted box appears more chaotic, which is very similar to the behavior of a non-cohesive class. The match between size and WMC is rather obvious because the concept of code size and box size are naturally related in everyone's mind.

4. PROGRAM REPRESENTATION

4.1 Layout Techniques

There is no natural way to distribute all the elements of a software system on a plane. Geographical information systems (GIS) use maps to represent certain variables concerning a given territory [13] (an example is shown in Figure 2). Similarly, we decided to develop a map representation for a system. Software architectural information provides a good way to construct separations equivalent to country borders, states, cities, etc. Moreover, architecture represents valuable information on the quality and for the understanding of software.

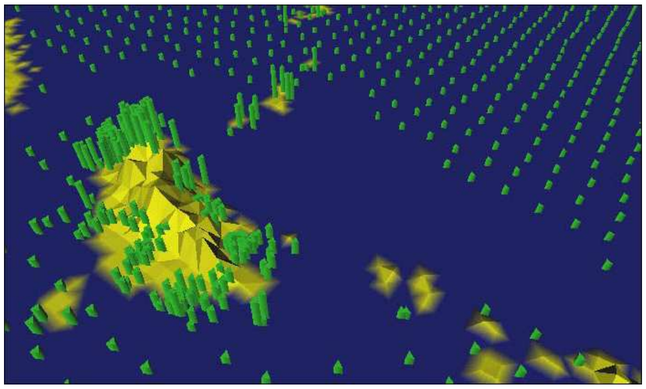

Figure 2: GIS representation: Typhoon conditions across Southeast Asia during Summer 1997 [9].

Classes are included in packages that may also be included recursively in other packages. These hierarchical groups of elements can be separated in areas and therefore, simulate a form of geographical map. This configuration helps to identify which portions of code display abnormal values or amounts of a given characteristic. We have currently developed two different types of class layout: *Treemap* and *Sunburst*. The details on these two layout techniques follow in the next sections, and an experiment evaluating their respective efficiency is described in Section 6.

4.1.1 Treemap

Treemap was introduced by Johnson and Shneiderman [10] to visually represent a file system. This was an important tool to visualize offenders in the recurrent problems of disk space shortage common at that time. The original Treemap starts with a rectangle that represents the root of a hierarchy. This rectangle is split in a number of vertical slices equal to the number of its children. Each slice has a width proportional to the size of its node. These new rectangles (slices) are then split horizontally the same way. The splitting process goes on, alternating between vertical and horizontal separations. This algorithm, called slice-and-dice, is illustrated in Figure 3.

The Treemap representation must be adapted to a representation suitable for software structures. Indeed, Treemap uses continuous values while because our software basic element is the class with a discrete representation, we need to allow each entity to appear without interference. Therefore regions of space have to be an integer factor of this basic element spatial requirements, horizontally as well as vertically. This spatial requirement corresponds to the distance that must separate two elements (3D boxes) of the largest observed dimensions (sizes). The size of a node corresponds to the total number of classes contained in its sub-tree.

Our solution splits the Treemap with the original slice-and-dice algorithm, but rounds up the number of elements in order to fit in a rectangular slice. During the recursive construction of each sub-node, it may be impossible to fit its representation within its allocated rectangle. This occurs when many subdivisions cause a difference in the amount of

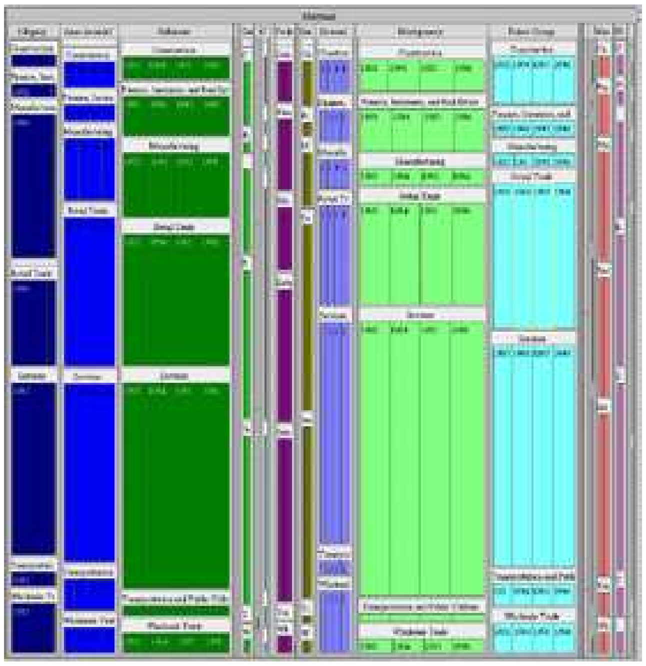

Figure 3: Original 2D Treemap representation of census data.

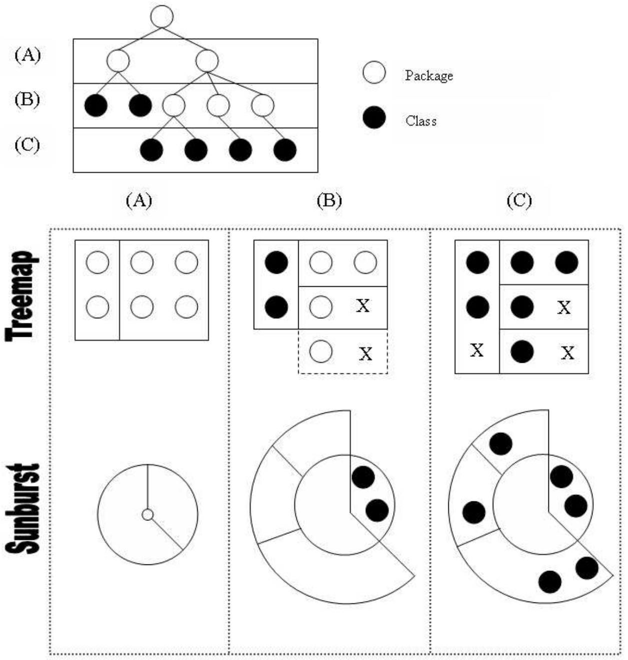

Figure 4: Illustration of converting a software hierarchy in our adapted Treemap and Sunburst techniques.

space needed. We then extend the original rectangle possibly in both directions to respond to the increased required space, and refit these elements in the new increased rectangle. This is illustrated in the middle of Figure 4. When determining the original subdivisions in (A), the classes fit in the two rectangles. However when traversing the hierarchy in (B), the new subdivisions cannot be entirely filled with classes, and therefore new space is required. The original rectangle is then extended in (C) to accommodate for the new distribution of elements, but empty spaces (marked with X) are created.

In order to reduce the problems related to these empty spaces, we search for the perfect solution, considering we have a discrete treemap. We scan all possibilities at each level of the hierarchy and keep the arrangement that minimizes the number of holes. We get an imposed size from the upper level and then try a size in the opposite direction at the current level (which becomes the imposed size for its sub-level). This is obviously done in exponential time. Although, we added a few simple changes such as caching of solutions for a given pair (node,size) and a dichotomist search when trying sizes on current level. We managed to reduce the processing time to under 8 seconds for a system of 5000 classes. This takes 51 seconds on the 10000 and more classes of Eclipse. The large number of sub-packages are more to blame than the number of classes in this case. Although the number of holes seems to decrease significantly, the user overall feeling is rarely affected by the reduction.

An example of our Treemap subdivision is presented in Figure 5 (top).

4.1.2 Sunburst

We also adapted another layout technique to represent the architectural properties of a software system. The principle is the same as above. The space-filling algorithm is inspired by *Sunburst*, introduced by Stasko [17] (Figure 6). This algorithm builds a circular distribution of a hierarchy, and its primary purpose was again to visualize large file systems in 2D. It separates sibling nodes radially (by angles), and levels in the hierarchy by arcs at distances from the disk center. A navigation system with interactive zooming was also developed in the original tool.

As with our adaptation of the Treemap algorithm, we fill the circular space covered by Sunburst with class representations. Two classes must be separated by at least the basic distance necessary between two representations of the largest 3D box. Radial separators are used to graphically divide sibling packages.

As an example, starting from the root of the system architecture, assume a package contains 10% of all classes in the system. A 36-degree slice is thus allocated in the Sunburst circular representation. All the classes at this package level are distributed in the slice along an arc and then along the increased radius. If there are sub-packages in the package, an arc separator is drawn to separate the sub-packages from the classes, and radial separators are drawn according to the number of classes each sub-package contains.

The bottom of Figure 4 explains how a simple hierarchy is represented. In (A), the disk is divided in two slices, one for 2 classes (120 degrees) and one for 4 classes (240 degrees). In (B), the two classes are distributed along its corresponding arc. In (C), 3 sub-packages are first generating two radial separators, and the 2, 1, 1 classes are distributed along their respective arcs.

A result is illustrated in Figure 5 (bottom).

4.1.3 Holes in Both Representations

Unfortunately, our adaptations to both Treemap and Sun-

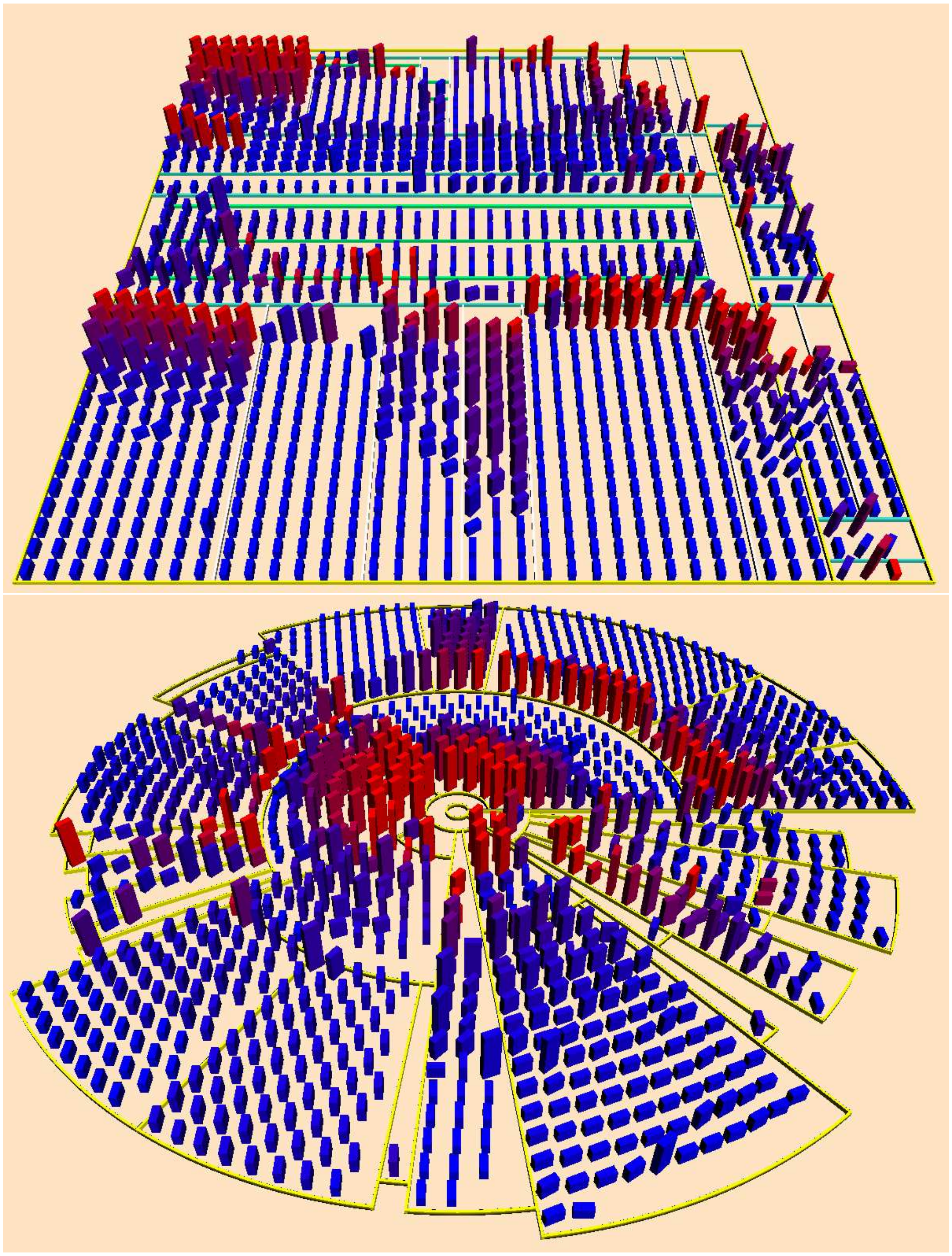

Figure 5: (Top) Modified Treemap technique and (bottom) modified Sunburst technique as displayed in our framework. They both represent *PCGEN*, a tool for character generation in *RPG* (1129 classes).

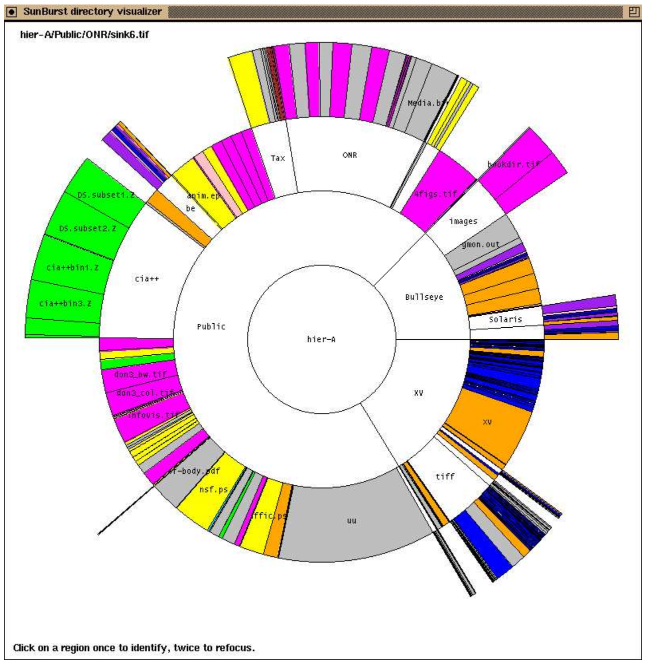

Figure 6: Original 2D Sunburst representation of a file system.

burst can still result in the insertion of holes in the representations. In simple illustrations such as in Figure 4, this looks like a serious problem. However, our framework is designed to study systems made of hundreds and thousands of elements. In fact, in the average cases (real softwares) that we tested, it did not prove a problem at all in our visualization. Figure 5 is very representative of typical softwares we studied.

4.2 Navigation

Although, our system is designed for immediate detection of patterns and structures, navigation through the graphical entities associated with the software metrics proved very useful to investigate details that may not have appeared otherwise. Navigation is also important to prevent mutual occlusions between opaque 3D boxes, since we operate in a 3D world. Our camera model is general enough to provide experts as much freedom as possible: it can move in any direction on and above the plane, it can zoom in and out, and can change the field of view. The camera also uses a few constraints to improve efficiency: it is constantly directed to the plane where the graphical entities are displayed, and it can be rotated on an hemisphere around any selected viewpoint.

4.3 Filters

For some analysis tasks, it is important to focus on a subset of elements while keeping a global view. We will see an example of such an analysis task in the following section. In this perspective, we use filters to put emphasis on useful elements, or to reduce the visual importance of useless elements, for instance, by removing their color. Two different categories of filters have been implemented. The first one deals with the distribution of the metric values. For example, we can focus on classes that have extreme values for a particular metric by changing their color to red and by giving the color green or yellow to the others (Figure 7). Extreme values are detected using classical statistical techniques such as plot boxes. The second category of filters exploits structural information. In addition to metrics, our environment allows to extract UML relations such as associations, aggregations, and generalizations. For a particular class, the expert can view only classes that are related to it by a particular type of link (Figure 8).

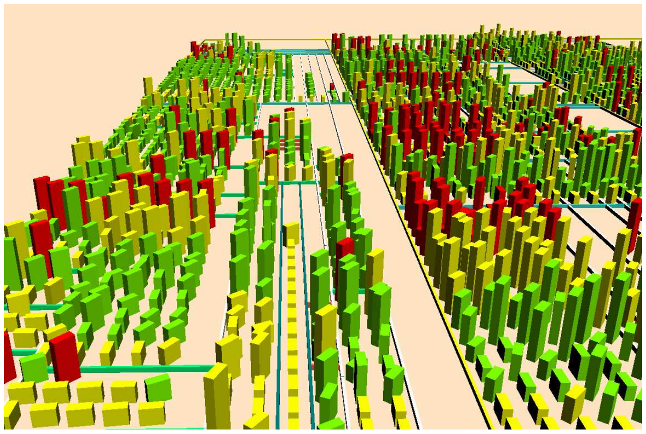

Figure 7: Extreme values for CBO are presented in red. It displays a part of *JRE* 1.4.04 from Sun Microsystems.

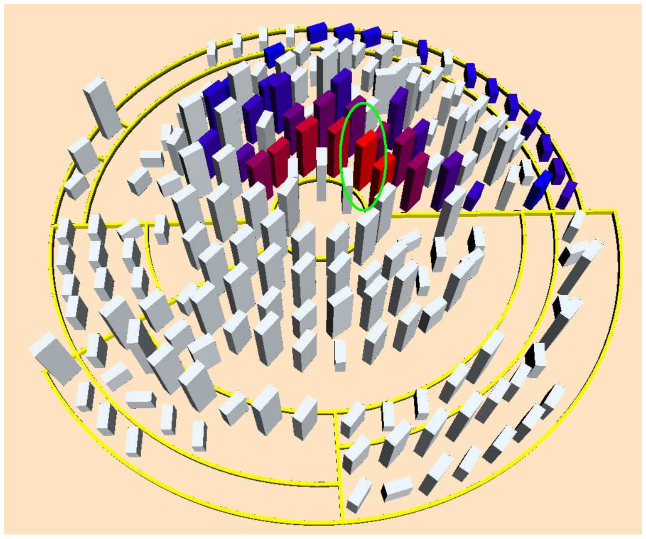

Figure 8: An example of the association filter: Only classes associated with the class circled in green remained in color.

5. ANALYSIS TASKS

In order to study the effectiveness of our framework, we tested it on three categories of software analysis tasks: detection of design principle violation, architecture understanding, and evolution analysis. We briefly present each category in the following sections.

5.1 Design Principle Violation Detection

One of the most well-known and important principle in quality analysis is the fact that code should always exhibit low coupling and high cohesion. However, this fundamental principle is very difficult to verify because finding threshold values and trade-offs between coupling and cohesion is very context-dependent. Using our framework, an expert can estimate whether a portion of the code violates this principle by taking into account the global context of the program. The violation can be detected at the class, package, and program levels without a need for aggregating the data (averages, median, etc.). Figure 9 shows an example of such a situation in our framework.

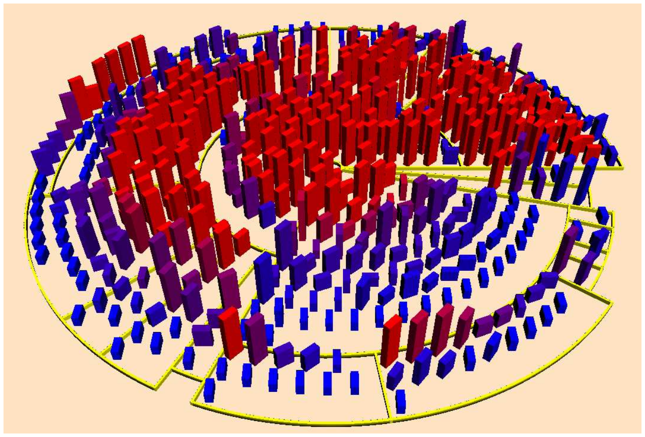

Figure 9: The predominantly red appearance indicates that the coupling in this program seems to have grown out of proportion. Maybe all this coupling is not essential. It represents *ArtOfIllusion*, a full featured 3D modeling, rendering, and animation studio (523 classes).

Another interesting analysis in this category is the detection of anti-patterns. Anti-patterns are known to be bad coding practices that may cause problems in subsequent development phases. An example of anti-pattern detectable by our framework is the *Blob* [3]. A Blob is an enormous accumulation of code in very few classes containing many complex methods. This anti-pattern is often caused by object-oriented code used in the context of procedural needs, or from inexperienced developers. In the context of the mapping proposed in Section 3.1, a Blob can easily be spotted in our framework. It appears as a twisted and tall box linked to small boxes. A Blob can be detected by applying a filter that reveals classes with an abnormal size-complexity value. Then, when one of these classes is selected, we apply on it a filter that identifies the classes related to it. If these related classes are all small, there is a high probability that we found a Blob. The detection can be more efficient if we modify the mapping with the DIT metric. Indeed, classes playing a role in the Blob anti-pattern are generally not deep in the inheritance tree. By associating colors to the DIT, it is easier to focus on classes wiht a small DIT value.

5.2 Functional Architecture Understanding

A graphical representation of a system is a good cognitive support to the understanding of programs. Indeed, the role of each class in the program has an impact on its metrics. For example, kernel packages contain a large proportion of complex classes with high coupling. In the mapping of Section 3.1, these classes are large and red. Similarly, most utility packages contain a large proportion of complex classes with medium-to-low coupling (large and purple). Without any additional (semantic) information to code, the expert can glance at our representation and have a quick evaluation of the vocations of the packages, which can ease his understanding. In Figure 10 for example, the package *PcGen.core* contains mainly red, large, and twisted classes. This gives an indication that this package is the kernel of the program, which is the case for *PcGen*. In the same example, *PcGen.Gui.Editor* contains many blue/purple, large, and twisted classes. Utility packages have usually the same shape. Finally, *Gmgen.plugingmgr.messages* contains a majority of blue, small, and straight classes. Classes defining types correspond to this description. In the context of *PcGen*, this package contains message types.

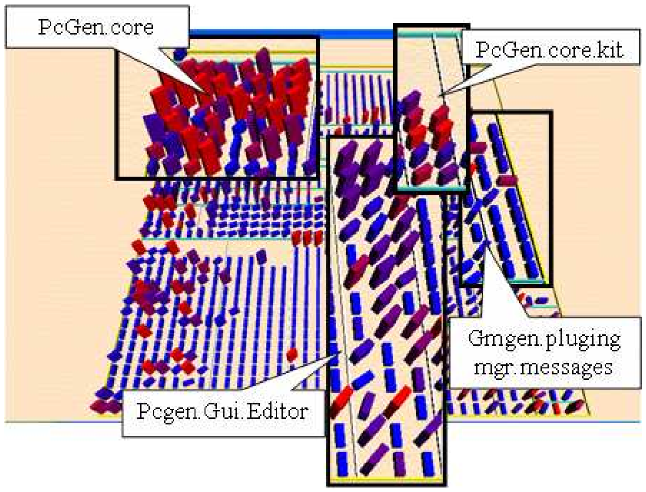

Figure 10: An example of the correspondence between visual patterns and package types.

5.3 Evolution Analysis

An important contribution of our framework regards software evolution analysis. We can analyze the evolution of a single class as proposed by Lanza and Ducasse [12], or package/program evolution. In the first case, we can observe the evolution pattern of a class and deduce its next evolution stages. For example, some evolution patterns are synonymous to dead-code classes. In the case of package/program evolution, we can observe the representation of multiple versions of the same package/program (see for example Figure 11). The evolution can reveal low quality packages and determine when a major refactoring is needed.

6. EVALUATION

To evaluate some aspects of our approach, we conducted an experimental study. We describe the elements of this study in the remaining of this section.

6.1 Experiment Objectives

We mainly target two objectives. First, we evaluate if the time needed to perform some analysis tasks compares

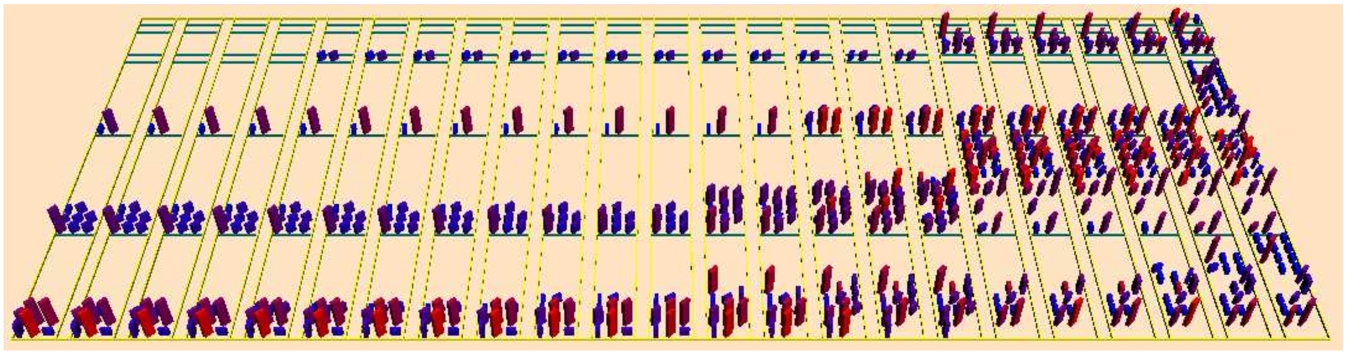

Figure 11: Representation of the evolution, from left to right, of a package over 23 versions. It represents *Quantum*, a database access plugin for *Eclipse* (689 classes).

favorably to using today's popular tools. This can also help us identify weaknesses in our framework.

Second, we will assess the effectiveness of layout techniques. Does sophisticated layout ease analysis tasks compared to a naïve layout? If so, which layout should be used in which context (Treemap *vs.* Sunburst)? Indeed, the design of efficient layout techniques is time consuming. While our proposed adaptations appear satisfying, there is still room for improvements, mainly to better exploit the entire display space, while preserving architectural information. However at this stage of our research, we need to verify if the effort needed for this improvement is justified.

The objective of this study is not to rigorously compare our visualization framework to other frameworks or to other analysis techniques. Although such a comparison is suitable, it is difficult to conduct a controlled experiment considering the diversity of the analysis tasks supported by existing tools. However, such an experiment remains one of our future objectives.

6.2 Experiment Settings

In this experiment we use three layout techniques: Treemap, Sunburst, and for comparison purposes, a naïve layout technique called *Treeline*.

Treeline represents classes of the architecture hierarchy with a depth-first algorithm. Each time we meet a node in the architecture tree, we place it on the current row, distributing the elements from left to right. Levels and packages are determined with separators, with a given separator color for each level of the hierarchy. So when the color switches, the following classes are in a different level. When the separator color remains the same, the next package is a sibling. This algorithm has an optimal use of the space and is easy to implement/execute. An example of this layout technique is presented in Figure 12.

The experiments are run in the form of an electronic questionnaire with 20 analysis tasks. Two types of knowledge are needed to perform the tasks: class characteristics and program architecture. 5 tasks involve exclusively the first type of knowledge, 5 tasks involve the second type, and 10 tasks involve both types. Task definition is inspired by the types we described in Section 5. For example, one task is to identify large packages that contain almost exclusively highly cohesive classes. Each task had to be performed on a different program, taken from different application domains, and with sizes ranging from 72 to 1662 classes. For instance, the task given as training used the associated program *JDK* 1.1 (1662 classes).

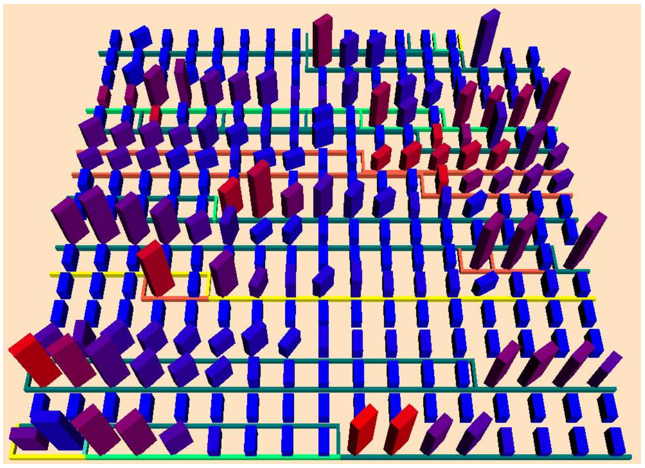

Figure 12: Example of the Treeline algorithm. It represents *EMMA*, a tool for measuring coverage of Java software (286 classes).

The time taken by a subject to correctly perform the task is automatically recorded. In addition to the 20 tasks, subjects were asked questions about their subjective rating of the layout techniques. We used 15 subjects (graduate students) divided into 3 groups of 5 subjects. For each analysis task, subjects of each group are asked to perform the task using one of the three layout techniques (Treemap for group A, Sunburst for group B, and Treeline for group C). To avoid fatigue and learning effect biases, the assignation of a layout technique to a group is random and changed with the tasks. The order of tasks for subjects from the same group is also random. For each pair (task:layout), we computed the average time of the 5 subjects. We used the computed value to rank the techniques for each task. Then, for each layout technique, we calculated the average and the median for all tasks.

Subjects are volunteers. Their motivation should not be biased by any form of evaluation. Most of them are software engineering researchers. To avoid significant differences between them, they received a quick training on the environment before the experiment. They learned how classes and packages are represented within each layout technique, how to navigate in this 3D space, and finally how to perform the tasks and record time.

6.3 Results

At the end of the experiment, we obtained 300 time entries (20 tasks $\times$ 15 subjects). All the subjects gave correct answers within the allowed time. The compiled results are presented in Table 1.

	Sunburst	Treemap	Treeline
Average Time (sec.)	50.13	36.38	69.88
Median Time (sec.)	23.87	22.24	41.24
Best Time	35%	55%	10%
Preferred System	80%	13.33%	6.66%

Table 1: Experiment Results

As expected, the average and median times confirm that our visualization framework allows to perform relatively complex analysis tasks on small-to-medium size programs in less than one minute. For example, it took 50 seconds on average to identify large cohesive packages in a program with 1662 classes (see Section 6.2). On one hand, performing the same task manually using only code and tables of metric values would probably take significantly much more time. On the other hand, the solution of writing a program that executes this task poses two problems. First, it is difficult to determine what *large* and *cohesive* means without having a way to appreciate the context. Second, even if this is possible, an expert has to write a program each time he wants to investigate a new analysis task.

The second interesting finding in this experiment is that sophisticated layout techniques play an important role in the analysis tasks. Indeed, both Treemap and Sunburst have an average time lower than the one for Treeline. The difference is more significant when we consider median values (reduction of almost 50%). Treeline obtained the best time only twice over 20 tasks. When we examined these two tasks, we noticed that in both cases, the classes/packages of interest were displayed immediately in the middle of the representation. This gives an advantage to Treeline, knowing that for these two cases the situation was completely different for the other two layouts.

The other objective of the experiment is to determine which layout technique is the most useful. Our results revealed that Treemap and Sunburst have almost the same average and median times. However, Treemap seems to have a slight advantage; it was ranked as first for 11 tasks (55%) while Sunburst was ranked first for 7 tasks (35%). We tried to explain this difference. After looking at the task descriptions where sunburst was rated second, it appears that most of them involved cohesion, and by extension, twist (in this experiment, cohesion is mapped to twist). Sunburst has difficulty to deal with twist. In Treemap, the packages are represented as squares; any twist of a class is quickly perceived. This is not the case for Sunburst.

For the subjective rating of the questionnaire, all subjects enjoyed their participation to the experiment and mentioned that the framework is useful in quality analysis. Many suggested some improvements to the navigation. Regarding their preference for layout techniques, surprisingly, 12 of 15 subjects (80%) mentioned Sunburst. This result is significantly different from the one observed for the tasks. This is a little puzzling. A possible explanation is that Sunburst appears to many people as more aesthetic and harmonic although these qualities are not in our opinion useful in software quality analysis and understanding.

7. CONCLUSION

In this paper, we have presented a visualization framework for quality analysis and understanding of large-scale software systems. It exploits perception capabilities of the human visual system to help quality-model developers understanding software-related phenomena and quality analysts evaluating large and complex programs. Programs are represented using metrics. Moreover, structural information is considered to help experts focus on a part of the program by the mean of filters.

We claim that our semi-automatic approach is a good compromise between fully automatic analysis techniques that can be efficient, but loose track of context, and pure human analysis that is slow and inaccurate. Our framework can be used for targeted analysis tasks as well as for more exploratory ones, which is more interesting in the complex domain of software engineering.

The experiment we conducted showed that our class and program representations are effective for complex analysis tasks. The more sophisticated is a layout technique, the easier to perform are the analysis tasks.

We used our framework to visualize and analyze programs containing up to 10000 classes without lost of performance. However, for some programs, layout techniques do not have an optimal use of the display space. This case occurs when programs contain many packages with very few classes. Although these cases are rare, we are currently working on an improved version of these techniques. Another limitation of our approach is related to the evolution analysis. We cannot concurrently display multiple versions of large-scale programs. We are working on different possibilities to compare multiple versions. These possibilities include sliders and animation.

Finally, we are investigating the interpretation of metaphors to add a new level of knowledge in the analysis [8]. Metaphors are powerful means to transfer knowledge from a well-known domain to another one [15]. We are focusing on the city metaphor [16]. There are several similarities between a city and a program from the complexity perspective. Classes can be viewed as buildings and packages as districts. The box-based representation and layout techniques add to this similarity as observed in several figures in this paper. One objective of this investigation is to see if the shape of a class can determine its function in a program exactly as for a building in a city (houses, factories, business buildings, etc.). The same idea can be applied to packages. A package role can be determined by the role of the classes it contains. In a city, districts have vocations that derive from the nature of the buildings they contain (residential district, industrial district, business district, etc.). Early results appear promising.

8. ACKNOWLEDGEMENTS

The authors thank FQRNT and NSERC for their continuing financial support. All participants to the reported study deserve a special mention. Finally Karim Dhambri

contributed to this work with his implementation of filters in our framework.

9. REFERENCES

[1] D. Bell. *Software Engineering, A Programming Approach*. Addison-Wesley, 2000.

[2] L.C. Briand and J. Wuest. Empirical studies of quality models in object-oriented systems. In *Advances in Computers, 56*. Academic Press, 2002.

[3] W.J. Brown, R.C. Malveau, H.W. McCormick, III, and T.J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley Press, 1998.

[4] S.R. Chidamber and C.F. Kemerer. A metric suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):293–318, June 1994.

[5] A. Endres and D. Rombach. *A Handbook of Software and Systems Engineering*. Addison-Wesley, 2003.

[6] D. Hamlet and J. Maybee. *The Engineering of Software*. Addison-Wesley, 2001.

[7] N.E. Fenton and M. Neil. Software metrics: roadmap. In *ICSE - Future of SE Track*, pages 357–370, 2000.

[8] H. Graham, H.Y. Yang, and R. Berrigan. A solar system metaphor for 3D visualisation of object oriented software metrics. In *Australasian Symposium on Information Visualisation*, pages 53–59, 2004.

[9] C.G. Healey and J.T. Enns. Large datasets at a glance: Combining textures and colors in scientific visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):145–167, 1999.

[10] B. Johnson and B. Shneiderman. Treemaps: A space-filling approach to the visualization of hierarchical information structures. In *IEEE Visualization*, October 1991.

[11] C. Knight and M. Munro. Virtual but visible software. In *Proceedings of the International Conference on Information Visualisation*, pp. 198-205, July 2000.

[12] M. Lanza and S. Ducasse. A categorization of classes based on the visualization of their internal structure: the class blueprint. In *Proceedings of OOPSLA 2001 (16th International Conference on Object-Oriented Programming, Systems, Languages, and Applications)*, pages 300–311. ACM Press, 2001.

[13] A.M. MacEachren. *How Maps Work: Representation, Visualization and Design*. Guilford Press, New York, 1995.

[14] A. Marcus, L. Feng, and J.I. Maletic. 3D representations for software visualization. In *SoftVis '03: Proceedings of the 2003 ACM Symposium on Software Visualization*, pages 27–36, 2003. ACM Press.

[15] L. Mason. Fostering understanding by structural alignement as a route to analogical learning. *Instructional Science*, 32(6):293–318, November 2004.

[16] T. Panas, R. Berrigan, and J. Grundy. A 3D metaphor for software production visualization. In *Proceedings of the International Conference on Information Visualization*, pp. 314-319, 2003.

[17] J. Stasko and E. Zhang. Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations. In *INFOVIS '00: Proceedings of the IEEE Symposium on Information Visualization 2000*, pp. 57-68, 2000. IEEE Computer Society.