Visualisation and Analysis of Software Quantitative Data

Guillaume Langelier, Houari A. Sahraoui, and Pierre Poulin

Université de Montréal, CP 6128 succ Centre-Ville, Montréal QC H3C 3J7, Québec, Canada
{langelig, poulin, sahraouh}@iro.umontreal.ca

Position. In this position paper, we claim that automatic techniques are limited in the context of quality analysis. We present a semi-automatic approach for quality analysis based on visualization.

1 Introduction

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 [1, 5].

Object-oriented 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. Many reasons can explain these facts. The most important one in our opinion is that many phenomena related to software such as its evolution and its reliability are complex. There is very little theory explaining them. Having a good understanding and modeling of these phenomena are essential conditions 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". Today, we have a unique opportunity to empirically study these phenomena. Indeed, large sets of software data are available through open source programs and open repositories [4].

In fact, the idea of empirically studying phenomena related to software in not new. Many studies were conducted especially by the community of software measurement and quality. However, in spite of the many quality estimation/prediction models published in the literature, concrete applications in industrial contexts are very rare.

The success of the analysis of software data sets depends on the used analysis techniques. Automatic analysis techniques (such as statistic techniques and machine learning algorithms) are usually limited when studying phenomena with unknown or poorly-understood influence factors. This is the case of software evolution and reliability for example. We claim that hybrid techniques that combine automatic analysis with expertise are excellent alternatives to them.

A powerful example of hybrid techniques is visualization. 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 synonyms of phenomena occurrences. Preprocessing and presentation are very important to control the size of the studied data. Indeed, expertise cannot be effective when large scale systems are observed. For example, if we look to a UML class diagram containing hundreds of classes, it is very hard to have an effective analysis.

In this context, we propose a visualization-based approach for complex system analysis that circumvents the size problem by using perception capability of human visual system.

2 Visualization framework

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.

Our visualization framework is organized into two levels: software element (class in for example) and program levels. As our goal is to deal with large-size systems, we decided to focus on macro analysis (i.e. class as basic element). Many previous visual systems have concentrated on detailing classes into methods and variables. They offer a fine granularity view of software that is important. These view can be connected to our framework in order complete the analyses.

A crucial decision when building visualization environments is to determine which data to visualize and how. Too much data looses structural understanding and too little hides potentially important information. An image of cluttered data suffers from occlusions, and a badly distributed/organized data possibly hinders existing links. Finally, a visualization environment must exploit the natural skills of the human visual system to be successful.

More concretely, because of the intangible character of software [8], we choose to use abstract information derived from it such as its size, cohesion, coupling, etc. This data is mapped into graphical data such as color, shape, size, orientation, etc. that can be easily perceived by the human visual system.

2.1 Class representation

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 machines, 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 three dimensions. 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 significant. The more graphical attributes we introduced, the more bias 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 representation.

Representing a class with a 3D box is not very useful if no mapping exist between this box and the class as a piece of software. Metrics formidably links those two entities because they provide an interesting way to conceptually represent a class. The first reason that explains this fact is that metrics have quantitative values so they are easy to manipulate. Consequently it is possible to use more powerful statistics on them and, as the next section will point out, trans- formations of their values are straightforward. Secondly, the metric model is ideal for the primary goal of our research: the evaluation and comprehension of software quality.

In order to accurately represent a class we chose four characteristics that we thought were relevant to the study of software quality: coupling, cohesion, inheritance, and size-complexity. Many implicit or explicit software design principle involve these characteristics. For example, it is well accepted amongst the software engineering community that software should demonstrate a low coupling and a high cohesion. Size and complexity are also relevant to quickly spot important classes or to analyze whether or not a class is too complex and needs refactoring.

The chosen characteristics are captured throughout metrics. Many of them were proposed in the literature. For example, coupling can be measured using CBO, cohesion with LCOM5, inheritance by DIT and size-complexity by WMC [3]. In the remaining of this paper, we will use some of these metrics to illustrate our framework.

Now that we have determined what characteristics we want to represent and chosen the most interesting graphical features, we need to determine a correspondence function between the two sets. Hence, color twist and size can be respectively matched to CBO, LCOM5 and WMC metrics for example.

The mapping must take into account the adequacy between the features. Indeed, the color feature is a continuous linear scale in hue from blue to red. This means that classes with low CBO will have a blue color while those very coupled will show a flashy red color; an average CBO will result in some variant of purple. The twist takes values between 0 and 90 degrees of angle. Classes with a low LCOM5 (i.e., very cohesive) will be presented as very straight boxes while classes with a high LCOM5 will be lying horizontally. Finally, classes with high WMC will be presented as tall boxes and classes with low WMC will be presented as small boxes. Size and twist are

also continuous and linear. Three examples of class representations can be seen on Figure 1.

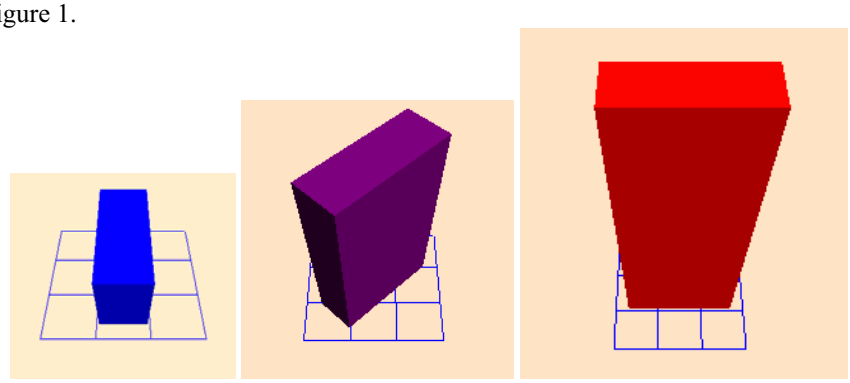

Figure 1. Three class representations. All three metrics; CBO, LCOM5 and WMC are increasing from left to right

In addition of providing good perception qualities, it also has some sort of semantic meaning. Indeed, a 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 the possible danger represented by that portion of code. Also, the twist is well suited for cohesion representation. Again, in collective imagination, the fact of being straight represents coherence and correctness. A twisted box would seem disoriented, 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.

2.2 Program representation

There is no natural way to place all the elements of a system on a plan instead of pure random. GIS (geographical information systems) use maps to represent certain variables concerning a given territory [10] (see for example Figure 2). We were inspired by this domain and realized that we should find a map representation for a program. The architectural information provides a good way to construct separations equivalent to country borders. Moreover it represents valuable information on the quality and comprehension of software.

Classes are included in packages that may also be included in other packages. It is possible to use these grouping of elements to separate areas and therefore, simulating a geographical map. This way, it becomes possible to tell what part of the code shows an abnormal amount of a given characteristic. We have currently developed two different types of class placement: the Treemap and the Sunburst.

The TreeMap has been introduced by Ben Shneiderman in 1991 [7]. At the time its purpose was to represent a file sys- tem. This was useful to solve the recurrent problem of disk space shortage common at that time. The initial Treemap starts with a rectangle that represents the root of a hierarchy. Then the root rectangle is split vertically in a number of slices equal to the number of its children. Each slice receives

a width proportional to the size of its node. These new rectangles are then split the same way by switching the separators to horizontal and so on (see Figure 3). This algorithm is called slice and dice.

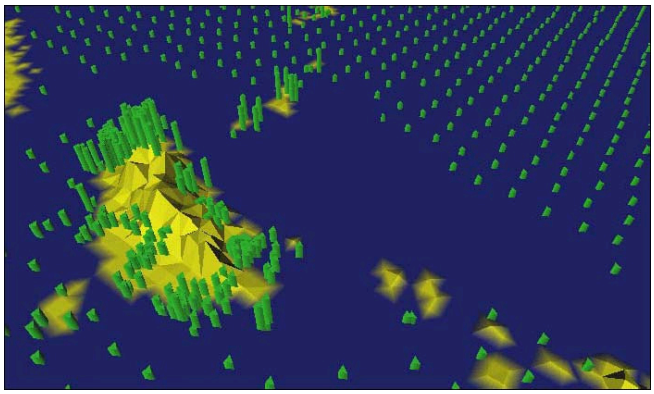

Figure 2. GIS representation: Typhoon conditions across Southeast Asia during the summer of 1997 [6]

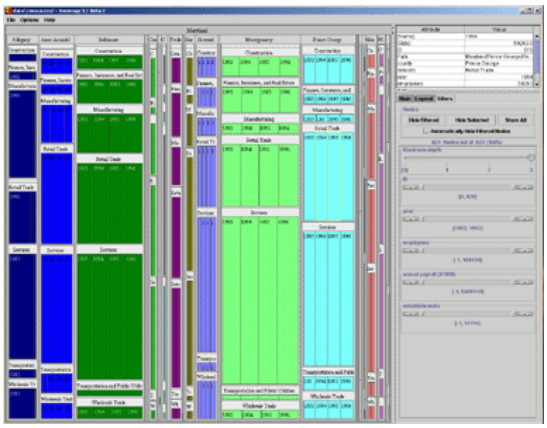

Figure 3. Original 2D Treemap algorithm representing census data

In order, to keep our box-based representation of the classes, we adapted the TreeMap algorithm by introducing the third dimension (3-D) and by splitting the space on the basis of the number of boxes to represent. Figure 4 shows an example of the application of the modified algorithm to PCGEN program (1129 classes).

We also developed another placement algorithm that reflects the architectural properties of a program. The principle is the same as for the TreeMap. The space-filling algorithm is highly inspired from the Sunburst algorithm introduced by John Stasko [11] (see Figure 5). This algorithm is a circular view of a hierarchy and its primary purpose is again the visualization of large file system in two dimensions. It uses angle to separate sibling nodes and length to represent the level in the hierarchy.

A navigation system with interactive zooming has also been developed for the original tool.

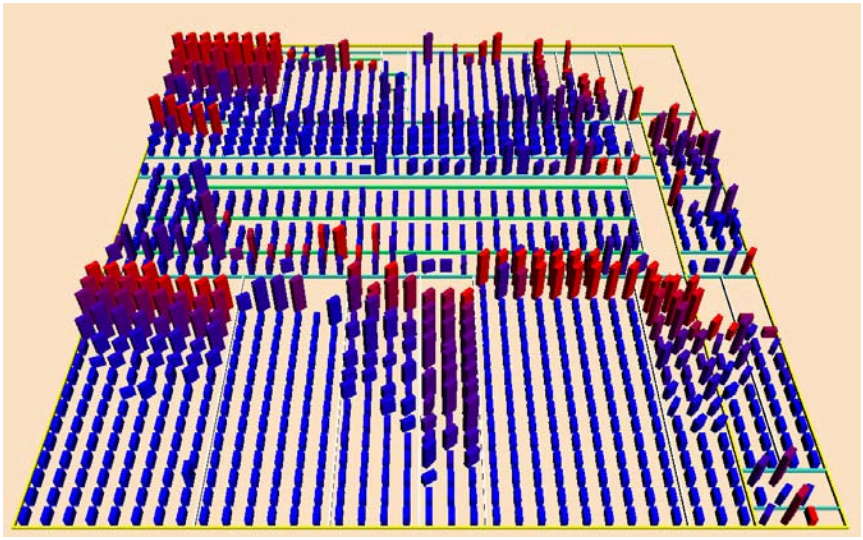

Figure 4. Modified Treemap representing PCGEN program (1129 classes)

This algorithm also has been adapted to our discrete needs and to the 3 dimensional world. We represent packages and sub-packages with arc portion but instead of coloring arc portions of a circle, we fill them with class representations discussed in Section 3 (see Figure 6).

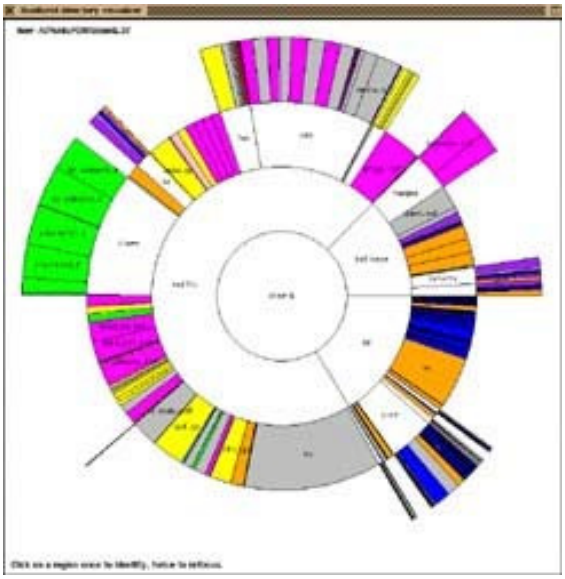

Figure 5. Original 2D Sunburst algorithm representing of a file structure

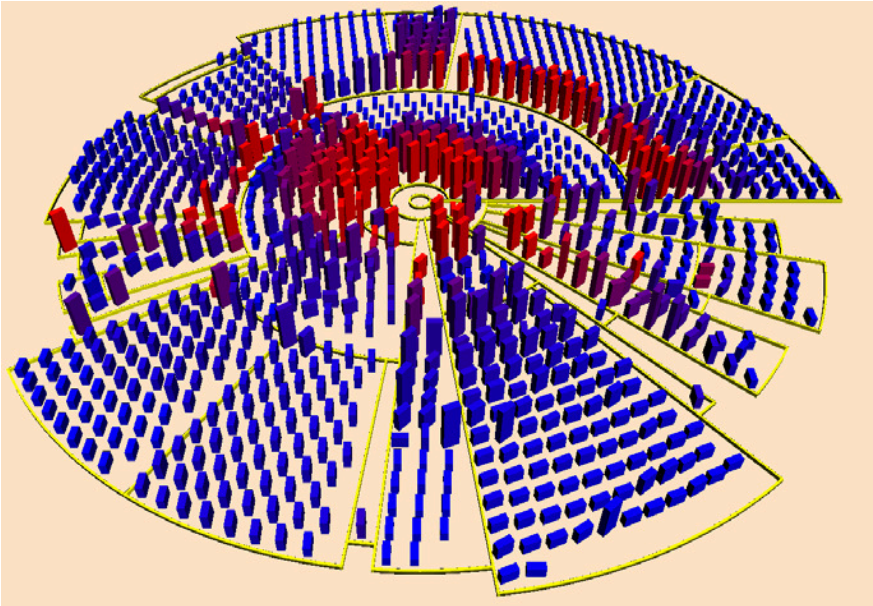

Figure 6. Modified Sunburst algorithm representing PCGEN program.

2.3 Navigation

Although, our system is designed for immediate detection of patterns and structures, navigation is useful to look on some details and to prevent occlusion since we operate in a 3 dimensional world.

Our navigation system gives the expert as much freedom as possible. The camera is constantly directed to the plan where the classes are drawn. It is possible to move the camera in any direction over the plan, to change the field of view angle by raising or lowering the camera and by zooming in and zooming out. It is also possible to move the camera on a semi-sphere around a view point. These commands seemed to be the best compromise between freedom and directed visualization.

2.4 Filters

For some analysis tasks, it is important to focus on a subset of elements when keeping the global context. We will see an example of such kind of analysis in the following section. In this perspective, we use filters that hide elements that are not useful.

Two different kinds of filters are 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 making the other classes transparent. Extreme values are detected using classical statistical techniques such as plot boxes. The second category of filters exploits structural information. Indeed, 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 (see Figure 7).

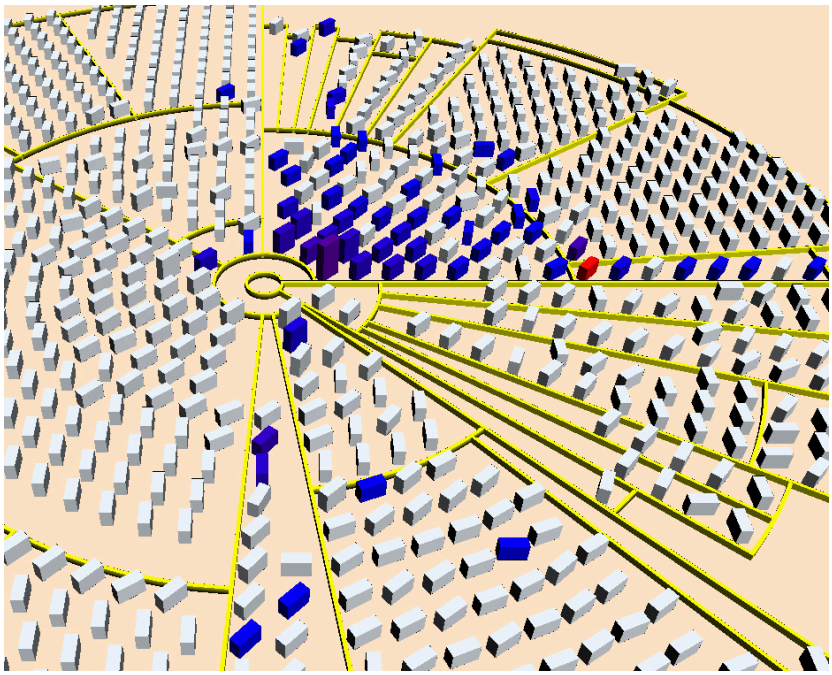

Figure 7. An example of the result of application of link-based filter to a class (big purple class in the center)

Filters can be combined into sequences that allow to perform complex analysis tasks.

3 Applications

We experimented our framework for three types of software analysis tasks. We will present each type briefly in the following sub-sections.

3.1 Design principle violation detection

As stated before, 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 hard to verify because of the difficulty of finding threshold values and tradeoffs between coupling and cohesion. Using our framework, an expert can judge whether or not 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 level without a need for aggregating the data (averages, median, etc.) Figure 8 shows an example of how the framework can be used in that particular case.

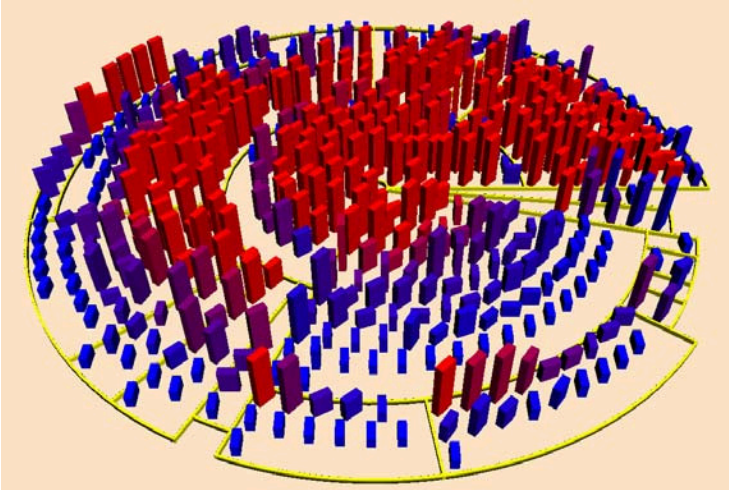

Figure 8. An example of program having a large proportion of coupled classes: ArtOfIllusion (513 classes).

Another interesting analysis in this category is the detection of anti-patterns. Anti-patterns are known to be bad coding practices that may result in problems in the subsequent phases of the development. An example that our framework is able to detect is the Blob anti-pattern [2]. The Blob is the enormous accumulation of code in very few classes containing a lot of complex methods. This anti-pattern is often caused by object-oriented code used in the context of procedural needs or when developers are inexperienced. In the context of the mapping proposed in section 2.1, a Blob can easily be spotted in our framework. It's a very red, twisted and tall box linked to small boxes.

Blob can be detected by first applying a filter that reveals classes that have an abnormal value for size-complexity. Then, when one of these classes is selected, we apply for it a filter that shows the classes that are related to it. If these later classes are small, there is a high probability that we are in a situation of a Blob.

3.3 Functional architecture understanding

A graphical representation of a system represents a good cognitive support to the comprehension of programs. Indeed, the role of each class in the system has an impact on it metrics. For example, kernel packages contain a large proportion of complex classes with high coupling. In the mapping of section 2.1, these classes are big and red. Similarly, most of utility packages contain a large proportion of complex classes

with medium-to-low coupling (big purple). Without any additional (semantic) information to code, the expert can have a first idea on the vocations of the packages which will ease the comprehension (see Figure 9).

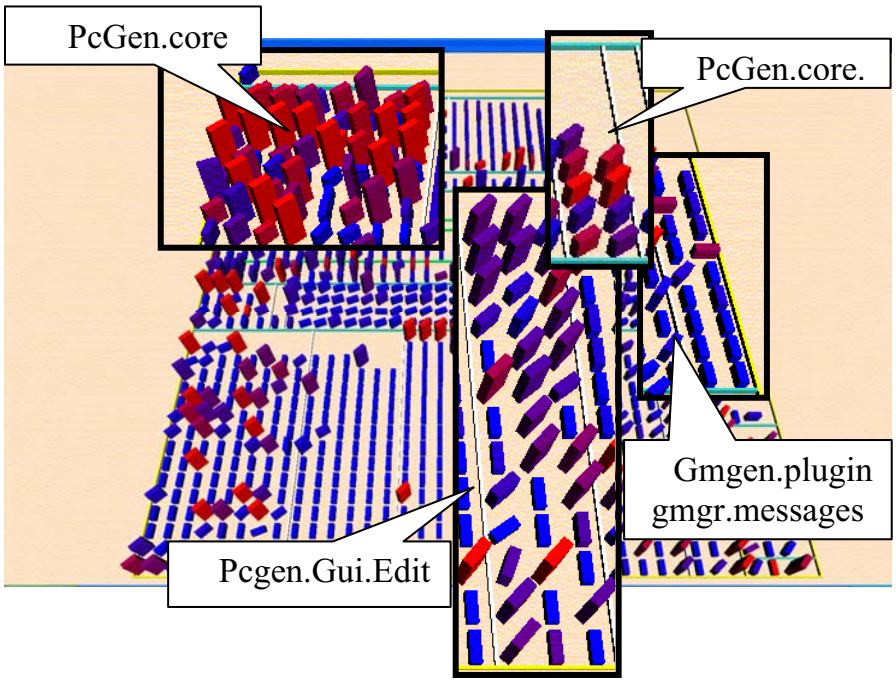

Figure 9. An example of the correspondence between visual patterns and package types

3.4 Evolution analysis

One important contribution of our framework is the software evolution analysis. We can analyze single class evolution as proposed by [9] 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 can be synonyms of 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 10). The evolution can reveal bad quality packages and determine when a major refactoring is needed.

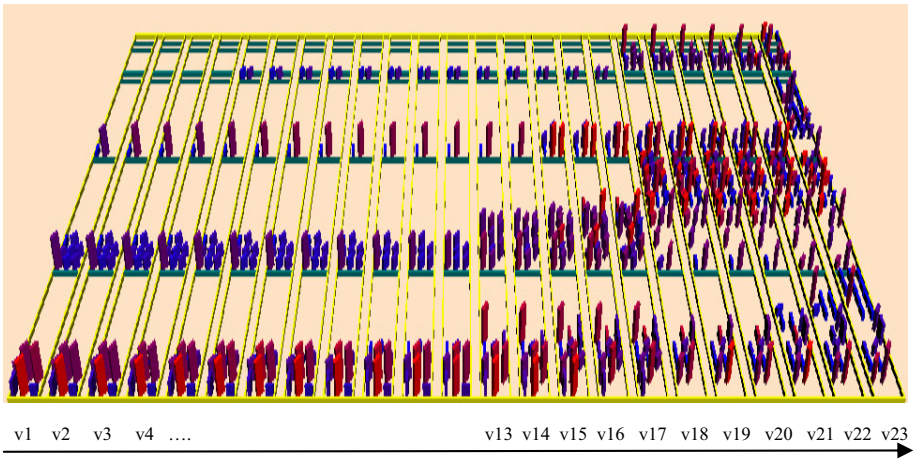

Figure 10. An example of a package evolution

4 Conclusions

In this position paper, we claim that for complex tasks of quality analysis, automated methods are limited and semi-automatic approaches are more appropriate. To illustrate our position, we have presented a visualization framework for large scale system using object-oriented metrics. It uses perception capabilities of an expert to evaluate the quality of a program. We briefly introduced some analysis tasks that can be performed using our framework.

Our semi-automatic approach is a good compromise between fully automatic algorithm that are efficient but loose track of context and the pure human analysis that is in ideal cases slow and in practice for large-scale programs inappropriate.

References

1. D. Bell. “Software Engineering, A Programming Approach”, Addison-Wesley, 2000.
2. 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.
3. S. R. Chidamber and C. F. Kemerer. A metric suite for object oriented design. In *IEEE Transactions on Software Engineering*, 20(6):293–318, 1994.
4. A. Endres et D. Rombach. “A Handbook of Software and Systems Engineering”, Addison-Wesley, 2003.
5. D. Hamlet et J. Maybee. “*The Engineering of Software”,* Addison-Wesley, 2001.
6. 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.
7. B. Johnson and B. Shneiderman. Treemaps: A space-filling approach to the visualization of hierarchical information structures. In IEEE Visualization Conference, October 1991.

8. C. Knight and M. Munro. Virtual but visible software. In Proceedings of the International Conference on Information Visualisation, 2000.
9. M. Lanza and S. Ducasse. A categorization of classes based on the visualization of their internal structure: the class blueprint. In Proceedings of 16th International Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2001.
10. A. M. MacEachren. *How Maps Work: Representation, Visualization and Design*. Guilford Press, New York, 1995.
11. J. Stasko and E. Zhang. Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations. In Proceedings of the IEEE Symposium on Information Vizualization, 2000.