Evolutional Insights from UML and Source Code Versions using Information Visualization and Visual Analysis

Shawn A. Bohner, Denis Gračanin, Troy Henry
Virginia Polytechnic Institute & State University
Department of Computer Science
Blacksburg, VA 24061, USA
{sbohner, gracanin, trhenry1}@vt.edu

Krešimir Matković
VRVis Research Center
Donau-City-Strasse 1
A-1220 Vienna, Austria
Matkovic@vrvis.at

Abstract

Risks associated with producing todays software applications are increasingly linked with size and complexity — just too many aspects of software to fit in the head of even the most talented software engineer. Understanding software entails more than browsing the source code or reviewing the models in the other software artifacts. We use information visualization and visual analysis techniques to parse data sets generated from UML and Java SDK source code versions to examine patterns. This visual perspective provides relevant insights and additional navigation opportunities for software engineers during development and maintenance activities.

1. Introduction

With the relentless growth in software, automated support for visualizing and navigating software artifacts is no longer a luxury. The sheer size and complexity of many of todays software systems exceed most human capabilities to grasp them. Experience over the years visualizing software (analysis, design, and code) has shown that there is often information that is obscured or lost in the abstractions used to render software visualizations [3].

Software visualization has long been used to seek insights into the structure and composition of software [4]. Software design and more recently architecture metrics have been gleaned both from source code and design artifacts of the development process [1].

An important aspect of this research is to identify the patterns of change present in the various software releases. Research has shown that many of the changes to software occur and reoccur in the same areas of source code. To investigate this, we downloaded and analyzed several versions of the JAVA SDK software as it evolved since it was first released. We were able to also download an early version of the Unified Modeling Language (UML) analysis and design models in Rational Model Definition Language (MDL). Our objective with these was to examine how the design faired against the strain of various releases.

We use a UML diagram to provide a starting point for exploring classes and their properties in Java SDK. The availability of large number of Java SDK source code versions (72 versions) provides an excellent opportunity to explore changes in the code using the UML design diagram as a reference point. A plethora of information visualization techniques and views is provided by the previously developed ComVis information visualization tool [6]. ComVis includes conventional views like 2D and 3D scatter plots, parallel coordinates, histogram, as well as other curve views used for displaying function graphs. It provides features like brushing, sorting, and linking that are very useful for visual analysis.

In this paper, we first examine software complexity in the context of todays increasingly large and complex systems. We then discuss the notion of software change and how visualizing it can help to understand the evolving system. We analyze the JAVA JDK and present out conclusions.

2 Related Work

Software complexity is the degree to which software (system or component) is difficult to analyze, understand, or explain [9]. Software visualization technologies are often used to illumine insights or patterns in the software artifacts the help us to better analyze, understand, or explain aspects of software systems; and in this way, visualization reduces the perceived complexities of software.

Information about software evolution can be gathered by analyzing a history of software releases. Software archives contain historical information about the development process of a software system. Standard visualization tech-

niques can provide us with an ability interactively explore that information [2]. We can apply data mining techniques in combination with information visualization to explore patterns and rules.

Software archives are usually implemented as source-control systems that preserve change-sets of files as atomic commits. If the specific order in which files were changed is not available, heuristics can be used to find sequences of changed files [5]. This approach provides the (unordered) sets of files with (partial temporal) ordering information.

In order to effectively use information visualization techniques, we need a well defined data model. A data model consists of a data definition and a manipulation language (structuring and operational definitions). The data model applies to all the data sets under consideration, described, explored, analyzed or manipulated.

Conventionally, data analysis approaches, such as traditional statistics or OLAP techniques [8], assume a relatively simple multi-dimensional model [7] (simple with respect to the separate data dimensions). For complex data sets it is necessary to provide an adequate data model.

3 Data Model

In our approach we are considering a two-level data set that consists of data points (tuple values) of n dimensions. The data set under consideration is $D = \{\mathbf{x}^1, \ldots, \mathbf{x}^i, \ldots, \mathbf{x}^d\}$ where d is the size of the data set and each $\mathbf{x}^i = (x_1^i, \ldots, x_j^i, \ldots, x_n^i)$, a collection of attributes, one for each dimension. A tuple attribute x_j^i can be categorical, numerical, or a data series itself.

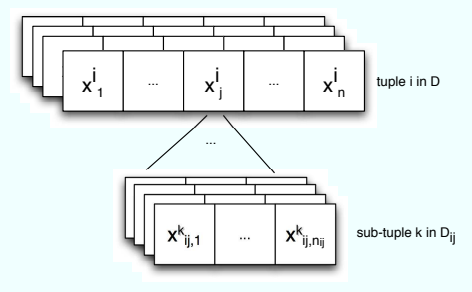

Figure 1. Generic data tuple

For each tuple $\mathbf{x}^i$ and each data series attribute x_j^i in a data tuple, we have a separate set of "sub-tuples" with its own cardinality and dimension. The set of sub-tuples is defined as $D_{ij} = \{\mathbf{x}_{ij}^1, \ldots, \mathbf{x}_{ij}^k, \ldots, \mathbf{x}_{ij}^{d_{ij}}\}$ where d_{ij} is the number of sub-tuples. A sub-tuple in D_{ij} has a form $(x_{ij,1}, \ldots, x_{ij,n_{ij}})$ where n_{ij} is a sub-tuple size and each sub-tuple attribute is either categorical or numerical. The sub-tuple $\mathbf{x}_{ij}^k$ is then $(x_{ij,1}^k, \ldots, x_{ij,n_{ij}}^k)$. One can consider a data set with more than two levels of tuples, however we are limiting this discussion to two-level data sets.

Since we have a complex data set consisting of scalars and time series data across various dimensions, we will use multiple linked views. The mixture of function graphs and scalars is particularly challenging. We have used a combination of the curves view and conventional views in [6] in order to analyze similarly structured data sets.

The curve view shows all function graphs at once. When combined with brushing techniques it can be used to nicely display curves in focus and those forming the context. We have also used transparency to depict curves density. We have successfully displayed and explored data sets containing more then 40,000 function graphs per dimension using the curve view.

4 Visualizing JAVA SDK

The first step in visualizing data is to create a a tuple collection for a data set corresponding to the UML data. Once the UML data is analyzed, the source code versions are processed to create tuples with values that are data series. The data set analyzed contained scalars and simple data series (curves). The curve view shows all function graphs at once. When combined with brushing techniques it can be used to nicely display curves in focus and those forming the context. We have also used transparency to depict curves density.

ComVis tool supports composite brushing so a user can combine brushes in an iterative manner. For more detailed explanation of analytical procedures and brushing mechanism see [6].

4.1 Static View

The UML data set provides the initial insight into the software set. The linked views are selected to explore "interesting attributes." The bottom view uses parallel coordinates to represent number of attributes, number of methods, number of static attributes, number of throws, inheritance level, number of nested classes, number of classes that return objects of the tuple class and the number of classes containing objects of the tuple class (Figure 2). The upper three views, from left to right are a 2D plot of number of methods versus number of attributes; a histogram of number of throws; and a 2D plot of number of throws versus number of attributes.

One can immediately see by selecting classes with the highest inheritance level (6) using a rectangle brush that all other attributes are very low. Similar information can be gathered from the upper three views where the brushed data (colored rectangles) indicate small attribute values and narrow ranges of values.

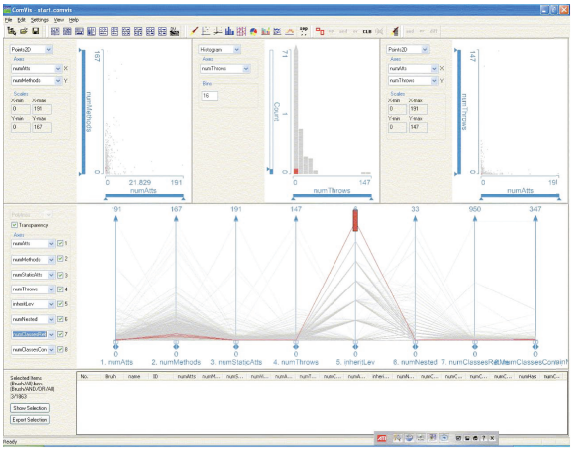

Figure 2. Visualizing UML reference data set.

As we move the rectangle down and select classes at lower and lower inheritance level, we see increase in the range of values of other attributes, as shown in Figure 3 (inheritance level 5) and Figure 4 (inheritance level 0)

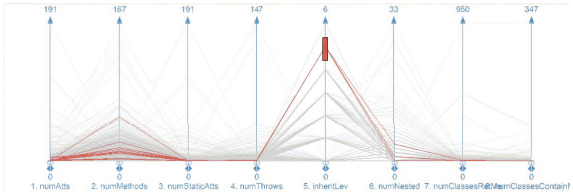

Figure 3. Inheritance level 5.

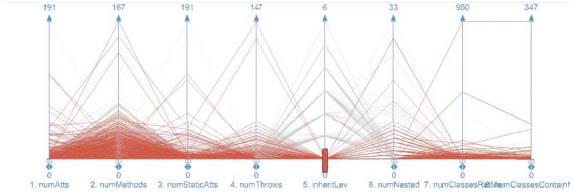

Figure 4. Inheritance level 0.

We can also explore other views to detect pattern in the data. The analytical procedure is iterative in nature. Incremental application and composition of brushing in the linked views provides focus on the relevant data patterns.

4.2 Multi-Version View

As the software evolves and changes, the attribute values change. The data set presented in the previous Section can be considered as a static snapshot in time. Access to source code versions allow as to gather many such snapshots and put them together using the described data model.

Each of the attributes now has a history and its value depends on the version of the source code. An individual, scalar value, is now replaced by a time series where the time dimension corresponds to the version number. Moreover, new data related to changes in the number of classes, packages and class hierarchy now becomes important.

In order to illustrate this point, Figure 5 provides an overall view of the Java SDK source code. The lower view provides a curve view [6], a family of 8,537 curves, each representing a class over a sequence of 72 versions (from 1.1.2_012 to 1.6.0). An individual curve has values 0 or 1, representing the change of appearance of a class, i.e. if it is present (value 1) or absent (value 0) in a give version of the code (time values from1 to 72). While a simultaneous presentation of 8,537 curves in a curve view is not necessarily useful, the ability to brush (select) to provide focus and content makes this view very useful.

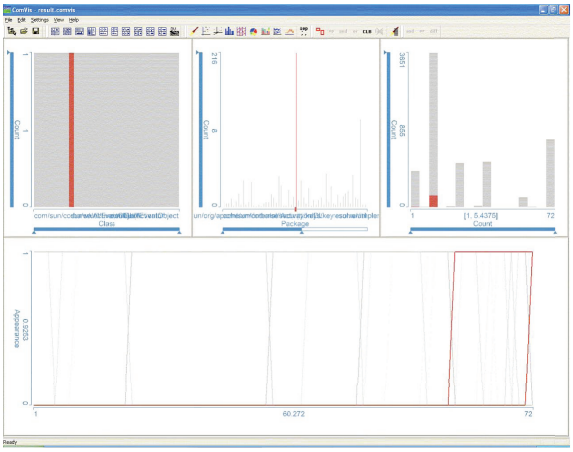

Figure 5. Time series example.

The upper three views, from left to right, include a histogram of individual classes; a histogram of number of classes in individual packages; and a histogram of number of classes that are present for a given number of versions.

For example, we can use the middle histogram to select a package that has the highest number of classes, 216, (com/sun/org/apache/bcel/internal/generic). We see from the left histogram the relative number of its classes compared to all the classes. We see from the right histogram that its classes are present in a small number of versions. Looking at the curve view we can see those classes (and therefore the package) appeared just recently.

The curve view provides other key information. There are six locations along x-axis where curves exhibit significant changes between 0 and 1. Those six locations correspond to the six major Java SDK releases, from 1.1 to 1.6. However, there are also several other locations with less visible change in appearance. Those indicate some smaller revisions. Using a curve view it is easy to detect and grade

changes in the code organization.

Another example of this type of visual analysis is provided in Figure 6. First we brush (1) all the curves that appear (change from 0 to 1) between versions 34 and 35 (Java SDK 1.3). The histogram of classes shows there are many new classes introduced here. Once can say that the curve view provides more qualitative picture while the histogram provides more quantitative picture.

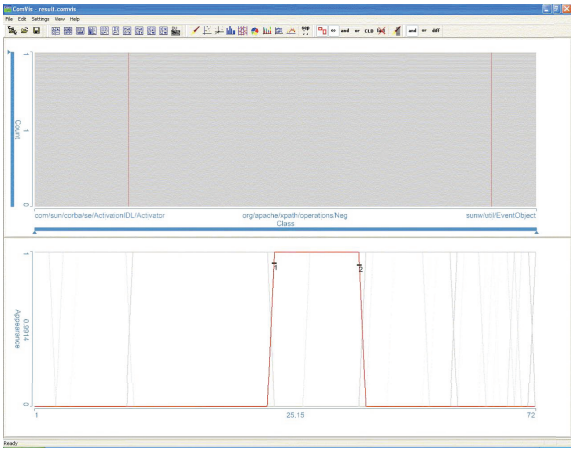

Figure 6. Classes appearing in JDK 1.3.

Now we add (logical AND) another brush (2) of all the curves that disappear (change form 1 to 0) between version 47 and 48 (Java SDK 1.4), we see that there are only two classes that appear only in Java SDK 1.4 (Figure 7), com.sun.java.swing.plaf.windows.WindowsMenuUI and org.apache.xpath.operations.Neg.

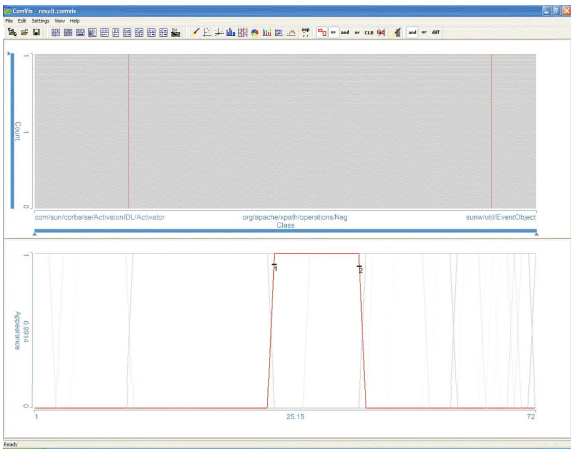

Figure 7. Classes appearing in JDK 1.3 and disappearing in JDK 1.4.

Switching between the static and multi-version view of the software provides additional level of visual analysis.

5 Conclusions

The analysis of relationships within a complex data se is a common task in many application domains, including software visualization. A novel combination of linked views, advanced brushing, and curves view represents a valuable tool for interactive visual analysis and analysis of data sets that include multiple families of function graphs (curves/surfaces).

The process of the composite brush construction captures the essence of interactive visual analysis procedures: It is interactive and iterative. The initial brush provides the initial data selection in one view. That selection is immediately displayed in the linked views and analyzed from different perspectives to formulate a hypothesis that is then tested using new brushes. During this iterative procedure, new, possibly unexpected patterns can be found.

This very general approach and tool can be effectively applied for software visualization to tackle the complexity of software evolution and changes. The key to that is to leverage existing UML model and/or source code versions and provide the available information using the described data model.

References

[1] B. Berenbach. The evaluation of large, complex UML analysis and design models. In *Proceedings of the 26th International Conference on Software Engineering*, pages 232–241, 23–28 May 2004.

[2] M. Burch, S. Diehl, and P. Weißgerber. Visual data mining in software archives. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 37–46, New York, NY, USA, 2005. ACM Press.

[3] B. George, S. A. Bohner, and R. Prieto-Diaz. Software information leaks: A complexity perspective. In *Proceedings of the Ninth IEEE International Conference on Engineering Complex Computer Systems*, pages 239–248, 2004.

[4] D. Gračanin, K. Matković, and M. Eltoweissy. Software visualization. *Innovations in Systems and Software Engineering: A NASA Journal*, 1(2):221–230, 2005.

[5] H. Kagdi, S. Yusuf, and J. I. Maletic. Mining sequences of changed-files from version histories. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 47–53, 2006.

[6] Z. Konyha, K. Matković, D. Gračanin, M. Jelović, and H. Hauser. Interactive visual analysis of families of function graphs. *IEEE Transactions on Visualization and Computer Graphics*, 12(6):1373–1385, Nov./Dec. 2006.

[7] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Elsevier, Amsterdam, 2006.

[8] E. Thomsen. *OLAP Solutions: Building Multidimensional Information Systems*. John Wiley & Sons, New York, 1997.

[9] H. Zuse. *Software Complexity: Measures and Methods*. Walter De Gruyter Inc, 1991.