A Flexible Approach to Visualize Large Software Products

Jean-Marie Favre
Laboratoire LSR-IMAG, Université Grenoble I
220, Rue de la chimie, Domaine Universitaire, BP53X
38041, Grenoble Cedex 9, France
http://www-adele.imag.fr/~jmfavre

Abstract

*There are so many kinds of software structures in a very large software product, that it is almost impossible to build a specific visualization tool for each specific need. In such a context, flexibility is very important. In this paper we claim, that producing a specific view on a large software product, should be as simple as using a spreadsheet to produce a new view on an arbitrary set of data. Instead of building visualization tools from scratch, existing components should be reused whenever possible. In particular it should be possible to connect interactively source components (those providing information on software) and visualization components (those displaying graphical views). To support this approach, we have built G^{SEE}, a **G**eneric **S**oftware **E**xploration **E**nvironment making it possible to visualize virtually any kind of software structures at a very low cost.*

1. Introduction

It is widely accepted that visualization can help in understanding software. However, most of the work done today is oriented towards *program visualization.* This is not surprising since (1) *programs* are the most common example of software artifacts, (2) *programs* are based on well-defined software models [20] (examples of entities include statement, variable, procedure, etc.; examples of relation include call, inherits, etc.). As a consequence a wide range of techniques have been proposed to visualize specific program structures resulting from a static or dynamic analysis of programs. Some of these techniques have been included in CASE tools, displaying for instance control flow graphs, call graphs, inheritance hierarchies or profiling information.

However *software visualization*, should not be reduced to *program visualization*. In particular, very large software products developed by very large teams are based on a wide range of concepts including for instance configuration parameters, libraries, licensing information, makefiles, software components, product lines, etc. Usually the set of software models used to manage large software products is not well defined. Moreover, these software models are often specific to a particular software company and evolve over time.

In a large software company, the usage of visualization techniques should not be restricted to programmers: they are plenty of potential users including for instance software architects, team managers, members of quality teams, configuration management teams, business teams, etc. These various stakeholders have many different needs and requirements on software visualization tools [21]. Unfortunately, currently they have to manage software entities in a “blind” way. Not because the information is not available, but *because of the cost of building a specific visualization tool for each specific situation.*

Tough a company can buy a tool to visualize programs written in a conventional programming language, there is no tool on the market to visualize large proprietary software structures in a flexible way. This is a pity because it is more difficult to understand large software products than small programs. Tough promising, many visualization techniques have failed so far to find their way to industry, in part because existing tools are not flexible enough to be adapted to a wide range of specific needs.

We believe that producing a specific view on a very large software product should be as simple as using a spreadsheet to produce a new view on an arbitrary set of data.

2. Visualization of data with spreadsheets

Using spreadsheets like Excel makes it very easy to generate pie charts or other graphical representations from arbitrary pieces of data. Indeed, this is probably the most popular application of visualization techniques today. We believe that analyzing the reasons of this huge success is interesting since a similar approach could be applied in the domain of software visualization. These tools make it possible to produce new views on existing piece of data, at almost no cost, by just specifying the data to be used and the result needed.

2.1. Getting basic data

With respect to the data to be used, spreadsheets are based on a *very simple data model* including elementary data types (e.g. numbers, strings, dates) and type constructors (e.g. table). These tools are totally *independent from the actual data* represented (as long as it can be represented in terms of the data model). They are also *independent from the actual sources of data*: data can be imported in a wide range of formats, directly accessed from a database, etc. *New data sources can be added interactively*: the information available for further processing results from the fusion of all data sources.

2.2. Deriving new information

The main power of the spread sheet approach is that *new derived information* can be added interactively thanks to a *high-level declarative language*. General-purpose operators include aggregate functions (e.g. sum or average), filtering operations, etc. Specific requirements can also be addressed via the integration of specific libraries (e.g. adding a library of statistical functions) into the environment. One of the strongest features of a spreadsheet is that it makes it possible to analyze existing data and produce new information in a very flexible way.

2.3. Creating new views

The production of graphical views from this data is another strong feature of spreadsheets and associated tools like report generators. Any piece of data can be visualized in many different forms by just *selecting a visualization technique* (i.e. a pie chart, a 3D histogram, etc.) and *setting appropriate rendering parameters* (i.e. binding the axis to a collection of values, the color to a given expression, etc.). The *view is specified in a declarative way* using the language described above extended with operators suited to visualization. Default values enable to get a first result very quickly. The view can then be improved interactively by refining the view specification.

An interesting aspect of modern spreadsheet environments is their extensibility. As point out before it is possible to integrate new sources of data, new libraries of operators. It is also possible to integrate new visualization components, to display for instance geographical information on a map of the world.

3. Visualization of software with GSEE

The reader may wonder how spreadsheets are related to software visualization. Indeed, we believe that the same approach should be followed, at least in the context of large software products. *It should be possible to create a new view on software at almost no cost*. As we pointed out before, there are so many kinds of entities involved in a large software products that a *specific approach* will fail: it is not feasible to build from scratch one specific tool for each specific view. A *generic approach* is thus needed.

To support this approach we define G^{SEE}, a *G*eneric *S*oftware *E*xploration *E*nvironment [9]. As we will see this environment is very similar in spirit to the approach presented above for the visualization of data. After all, software is just a specific case of data. Software visualization is a specific case of information visualization.

3.1. Background

The requirements for G^{SEE} have been based on our experience in building reverse engineering and visualization tools in non-traditional contexts including configuration management (e.g. [5,6,7]) and component models (e.g. [8,19,21]). In particular, the importance of a generic approach was further emphasized in the context of a partnership between the LSR laboratory and Dassault Systèmes (DS), the world's leader in CAD/CAM markets. Our study focused on CATIA, a large software product (1000 developers, 50 000 C++ classes; 1,000,000 files for all versions).

In this context we first built a specific visualization tool, called OMVT [21,9], dedicated to the visualization of software components developed by DS. The success of the first version of this visualization tool naturally brought up the problem of its evolution and generalization: interviews with software engineers revealed the existence of many other perspectives and software models within the company.

So we decided to follow a generic approach: we built G^{SEE} to be able to generate easily new views on demand, on arbitrary software models. G^{SEE} has been used successfully to visualize various structures of CATIA [9,21]. It is however totally independent from data sources and from visualization techniques. G^{SEE} follows the spirit of spreadsheets, but in the context of software.

3.2. Getting basic data

The first step to be able to manage arbitrary software structures is to define a suitable *data model*. Software structures are far more complex than those managed by spreadsheets. G^{SEE} relies on a simple yet powerful data model based at the end on the set theory. This model includes conventional types constructors such as sets, sequences but also functions [9] and relations. There is no limit on the type of entities managed, making it possible to deal indifferently with Statements, Variables, Configurations, Frameworks, etc.

Like spreadsheets, G^{SEE} is *independent from the source of data*. Since extracting information from software could be a very complex process, this function is encapsulated in *source components*. Source components are responsible to give access to some data extracted in some way from the software. The extraction of software facts is thus not the responsibility of G^{SEE}. However, providing easy ways to integrate source components is one of its responsibilities.

A common approach to integration is to store software facts in file exchange format such as RSF [18] or GXL [11]. Such file formats can be handled by G^{SEE} thanks to appropriated source components. However, the idea behind G^{SEE} is to look for information where it is, without unnecessary file conversion. For instance, like spreadsheets, G^{SEE} can directly access to data stored in databases[1]. Emphasis should be put on the fact that G^{SEE} is not restricted to access to components storing explicitly facts on software. On the contrary, one of its strongest features is its ability to handle source components computing on demand information on software. This allows for instance to directly access to parsers, cross-reference tools, profilers, a configuration management system, etc. What is more, new source components can be loaded interactively during a session. As an illustration, in [9] we show how JAssistant [2] is dynamically integrated within G^{SEE} to extract information on Java programs. Only five lines were necessary to get a first result.

3.3. Deriving new information

As pointed out before, the main power of the spread sheet approach is the ability to derive new information from existing one. The same applies to G^{SEE}. This environment provides a *high-level language to derive new information on software.* A wide range of operators is provided to deal with all the concepts available in the data model. This includes for instance operators like union of sets, sorting a set to get a sequence, filtering a collection of entities thanks to a predicate, etc. Just like a spreadsheet, G^{SEE} also includes some arithmetic operators and aggregate functions like sum and average. This makes it very easy to define and compute new metrics on software. G^{SEE} also includes more sophisticated operators like the transitive closure, operations on paths, etc. These operators are necessary to deal with complex structures of the software. As an illustration the following expression yields for a given package name, the names of packages this package inherits (directly or indirectly) (a class) from.

```
(package;getClasses;getSuperclass*;getPackage;getName)
```

This assumes that a source component extracting information from Java programs has been just loaded. The various identifiers are defined by the component. The ";" operator means the composition of functions. The "*" operator is the transitive closure.

It is also interesting to note that, like spread sheets, G^{SEE} allows to merge different sources of data in a transparent way, for instance data coming from a relational database with data obtained from plain files and composed with functions

1. This approach has been very useful in the context of the collaboration with DS, since software facts about CATIA were stored in Object-Store, an object oriented database system. This allowed to explore this software without writing any line of code!

implemented by a batch system. In the example above, the various identifiers are not actually defined by the same source component: different source components collaborate behind the scenes to give the result.

Like spreadsheets, the derived information has just the same status as basic information. So the model of software can be extended interactively. For instance the expression above can be used to extend the model, defining a new relation inherits between package entities.

3.4. Creating new views

The next step consists in creating new graphical views. Just like in the spread sheet approach, this should be done interactively. In particular it should be possible to choose the visualization technique and get first results very quickly. Then the view can be adjusted by changing rendering parameters, such as the mapping of color, etc.

To do that, G^{SEE} is based on the concept of *visualization component*. Such components are responsible to display information in some way. Currently, GSEE incorporates a large set of visualization components to display sequences, sets, tables, trees, treemaps, graphs, hypertexts, etc. Some visualization components were built from scratch in previous projects. For instance we built an implementation of tree map, as well as an implementation of sequence of lines inspired from SeeSoft [4]. G^{SEE} also gives access to third party components. This includes Swing component from Sun (JTree, JTable, JList, etc.), Grappa from ATT [10], etc. Emphasis should be put on the fact that the set of components is not limited: new visualization components can be loaded dynamically and integrated in the environment.

The information to be displayed by a visualization component is specified in terms of expressions described above. For instance, with a graph visualization component, the relation(s) to be displayed can be expressed as following: `getSuperclass | getInterfaces*`. This expression indicates that for each class, its super class will be displayed as well as all the interfaces it implements directly or indirectly.

With respect with rendering options, each visualization components is responsible to describe the parameters it is based on. The interesting point is that the value of these parameters are not restricted to constant (i.e. `background="grey"`): it is possible to use any expression of an arbitrary complexity. For instance, `height=getDeclareMethods#` maps the height of each node to the number of methods declared by this class (# means the size of a collection).

To alleviate the effort required to produce a new view, default values are provided. The G^{SEE} language also includes operators specific to visualization. For instance some operators are provided to convert an arbitrary range of values to a discrete set of colors, to convert an interval of values to a color gradation, to combine these operators, etc. What is more, default type conversions are automatically inserted when required. For instance, it is possible to write `color=getPackage` even if the `getPackage` function obviously returns a package entity, not a color. An arbitrary mapping will be chosen in this case and a caption will be created automatically, just like spread sheets do. The user can then improve the mapping if necessary.

4. Conclusion and further work

The current version of G^{SEE} has proved to be very useful in practice. This environment has been applied to a large set of software artifacts, including components, frameworks in the context of DS, bytecodes, statements, classes, packages, etc. in the context of java programs, workspaces, projects in the context of the Kawa Programming environment, historical information in the context of the Adele configuration management system, etc. Indeed, G^{SEE} is not limited to the visualization of large software product. For instance, we used it to display fine grain information: abstract syntax trees, byte-code structures, etc. However, the need for the generic approach is much more important in the context of large software because of the high number of entity types and relation types. What is more, since software models vary from company to company, it is not possible to build a tool from scratch. Actually, G^{SEE} is not only limited to software, tough some operators have been designed specifically to handle features typically associated with software structures.

There is a lot of space for improvement. Currently, the user interface is rather rudimentary. In fact, G^{SEE} is an object-oriented framework based on a component-based approach. Simple tools are provided for demonstration purpose.

While current components are passive, we are working on the integration of active components, sending events to G^{SEE}. This will allow for instance to animate views, with a minimal cost. For instance, we are planning the integrate active source components like JDPA to visualize the execution of Java programs. Similarly we continue our work on the visualization of software components, providing both static views and dynamic views. Finally, we are designing a component model on top of the JavaBean component model to get rid of its limitations. To goal is to be able to build sophisticated exploration tools, just by the interactive assembly of existing components.

5. Acknowlegment

We would like to thank Yann Dolisi and Remy Coulom who implemented the first version of TreeMap and LineSeq package of GSEE.

6. References

[1] CIA/++,CIAO http:// http://www.research.att.com/~ciao/

[2] A. David, "JavaAssistant 1.6, On-the-fly Class Browser", http://www.docs.uu.se/~adavid

[3] S. Demeyer, S. Ducasse, M. Lanza. "An Hybrid Reverse Engineering Approach Combining Metrics and Program Visualisation", Proc. of the Working Conference on Reverse Engineering (WCRE'99), IEEE, 1999.

[4] S.G. Eick, J.L. Steffen, E.E. Sumner, "Seesoft - A Tool For Visualizing Line Oriented Software Statistics", in IEEE Trans. on Software Engineering, Vol. 18, N. 11, Nov. 1992.

[5] J.M. Favre, "Preprocessors from an Abstract Point of View", Proc. of the Int. Conf. on Software Maintenance, Proc. of the Working Conference on Reverse Engineering, 1997

[6] J.M. Favre, "Understanding-In-The-Large", Proc. of the 5th International Workshop on Program Comprehension IWPC'97, 1997.

[7] J.M. Favre; "A rigorous approach to the maintenance of large portable software" in Proc. of the European Conference on Software Maintenance and Reengineering, IEEE, Mar. 1997.

[8] J.M. Favre, F. Duclos, J. Estublier, R. Sanlaville, J.J. Auffret, "Reverse Engineering a Large Component-based Software Product", Proc. of European Conf. on Software Maintenance and Reengineering, CSMR'2001, IEEE, March 2001.

[9] J.M. Favre, "GSEE: a Generic Software Exploration Environment", Proc. of the 9th International Workshop on Program Comprehension, IWPC'2001, IEEE, May 2001.

[10] Graphviz, http://www.graphviz.org/

[11] R.C. Holt, A. Winter, A. Schürr, "GXL: Toward a Standard Exchange Format", Tech. Report of Univ. Koblenz-Landau, May 2000.

[12] R. Holt et al, PBS: Portable Bookshelf Tools, http:// www.turing.toronto.edu

[13] Imagix, http://www.imagix.com

[14] Jinsight, http://www.research.ibm.com/jinsight/

[15] Kawa IDE. http://www.tek-tools.com/kawa/

[16] T.C. Lethbridge, J.Y. Pak, "Integrated Personal Work Management in TKSee Software Exploration Tool", Proc. of the 2nd Int. Symp. on Constructing Software Engineering Tools, CoSET'2000, June 2000.

[17] A. Mendelzon, J. Sametinger, "Reverse Engineering by Visualizing and Querying"

[18] H.A. Muller et al, RIGI, http://www.rigi.csc.uvic.ca/

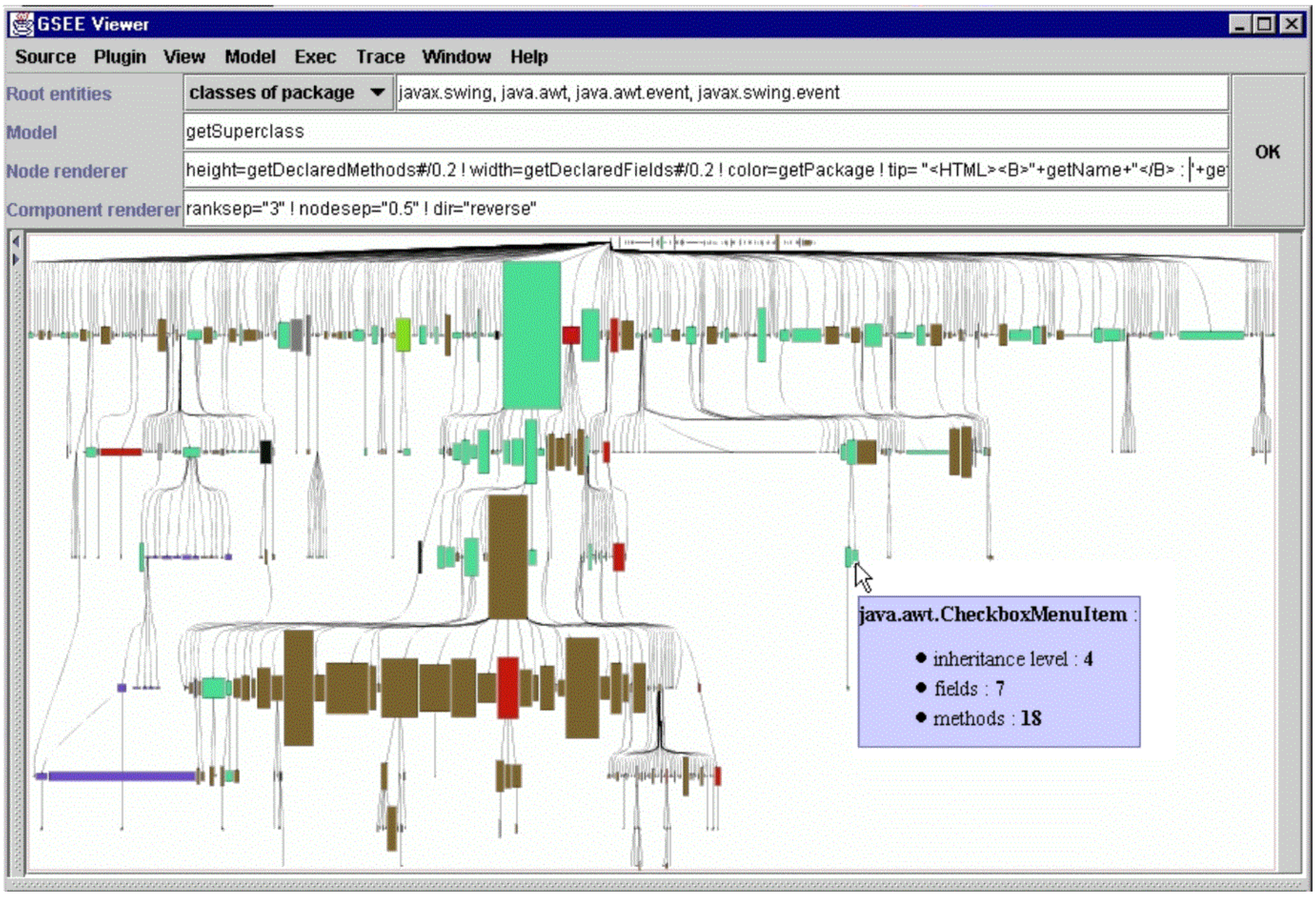
[19] S.T. Nguyen, J.M. Favre, Y. Ledru, J. Estublier, "Exploring Large Software Products", Proc. of ICSSEA, Paris, Dec 2000 (in french).

[20] D.E. Perry, "Software Interconnection Models", Proc. of the 9th Int. Conf. On Software Engineering, IEEE, March 1987.

[21] R. Sanlaville, J.M. Favre, Y. Ledru, "Helping Various Stakeholders to Understand a Very Large Software Product" Proc. of the European Conference on Component-Based Software Engineering, September 2001

[22] Sun, http://java.sun.com

[23] G. Sevitsky, W.De Pauw, R. Konuru, « An Information Exploration Tool for Performance Analysis of Java Programs », Proc. of TOOLS Europe 2001, March 2001.



Root entities: classes of package — javax.swing, java.awt, java.awt.event, javax.swing.event

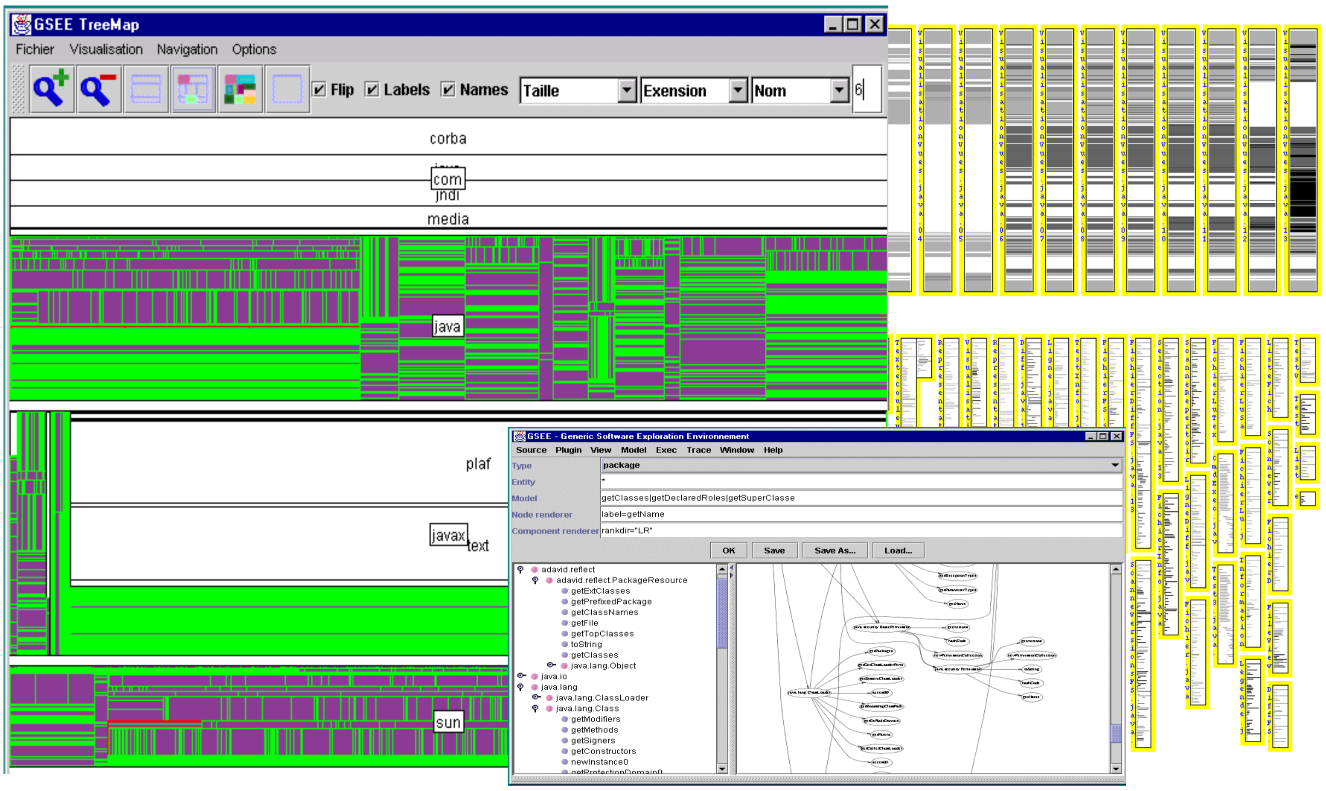
Model: getSuperclass

Node renderer: height=getDeclaredMethods#/0.2 ! width=getDeclaredFields#/0.2 ! color=getPackage ! tip= "<HTML><B>"+getName+"</B> : "+ge

Component renderer: ranksep="3" ! nodesep="0.5" ! dir="reverse"

java.awt.CheckboxMenuItem :

- inheritance level : **4**
- fields : **7**
- methods : **18**



GSEE - Generic Software Exploration Environnement

Type: package

Entity: *

Model: getClasses|getDeclaredRoles|getSuperClasse

Node renderer: label=getName

Component renderer: rankdir="LR"

```
adavid.reflect
  adavid.reflect.PackageResource
    getExtClasses
    getPrefixedPackage
    getClassNames
    getFile
    getTopClasses
    toString
    getClasses
    java.lang.Object
java.io
java.lang
  java.lang.ClassLoader
  java.lang.Class
    getModifiers
    getMethods
    getSigners
    getConstructors
    newInstance0
    getProtectionDomain0
```