

G^{SEE} : a Generic Software Exploration Environment

Jean-Marie Favre

*Laboratoire LSR-IMAG
220, Rue de la chimie, Domaine Universitaire, BP53X
38041, Grenoble Cedex 9, France
<http://www-adele.imag.fr/~jmfavre>*

Abstract

Large software products are very difficult to understand. One way to cope with this problem is to provide tools generating different software's views. Unfortunately, there are so many different entity types and relationships in a large software product that building a specific tool for each view is not cost effective. This paper presents G^{SEE} , a Generic Software Exploration Environment. G^{SEE} is made of an object-oriented framework and a set of customizable tools. Thanks to this environment, only few lines are needed to produce graphical views from virtually any source of data. G^{SEE} has been successfully applied to improve the understanding of different software artifacts including a multi millions LOC software.

1. Introduction

Large software products are very complex and therefore very difficult to understand. Several factors contribute to this complexity:

- **Number of entities and relationships.** The number of software entities and relationships clearly plays an important role during the comprehension process. For instance, a dependency graph at the module level contains several thousand nodes and edges. Understanding such a structure may be really difficult, though this graph is based on a single entity type (module) and on a single relation type (dependency).

- **Number of types and software models.** Complexity is also due to the large number of entity and relationship types involved. Complex software products may be based on hundreds of these types used to model concepts like variables, statements, components, versions, bug reports, test cases, libraries, product lines, etc. This large set of types can be structured into software models [23] ranging from programming models to configuration management models, architectural models, etc. In most of the cases, only the programming model is well defined, standardized and stable. Other models vary from company to company and evolve over time [8].

- **Diversity of software representations.** Eventually, every software artifact must have one (or more) concrete representations. The source code is the best-known software representation, but there are many other software representations including for instance Makefiles, DLL, Typelibs, configuration files, etc. Those software representations are often based on low-level languages or raw file formats: they are intended to be managed by specific tools or by only a few people with special skills. Besides, they often result from an incremental design with efficiency rather than understandability in mind [8].

- **Diversity of perspectives on software.** A perspective corresponds to a kind of activity undertaken by a software engineer playing a given role within the company. Large software products can be viewed from many different perspectives.

Simply put, large software products are difficult to understand because they are made of many entities of many different types in many concrete representations, usually not designed with software comprehension in mind.

A common approach to cope with understanding problems consists in providing the software engineers with reverse-engineering and visualization tools in order to reduce the cognitive effort [28]. These tools make it possible, for a given perspective, to extract a subset of the software entities, focusing on a subset of a software model. Visualization techniques are then used to produce suitable graphical representations. Software exploration tools enable software engineers to navigate between view or between perspective [28].

Many tools have been proposed (e.g. Rigi [21], PBS [13], CIA [2], Hy+ [20], Moose [25], CodeCrawler [5], Imagix [14], Spool [24], SeeSoft [6], TkSee [18], Dali [17]). Some techniques have been integrated into commercial CASE tools, including for instance call graphs or class inheritance browsers. The main body of work in software exploration can be qualified of “**specific approach**”: a specific tool is provided for each specific perspective. While effective in some situations, this approach is particularly weak in the context of very large software companies because: (1) only a few software models are covered by specific tools; (2) the cost of building a new specific tool may be prohibitive.

This results in situations where software companies store and maintain interesting information, they don't really use because the cost of presenting this information in an useful way is too high.

This paper concentrates on a “**generic approach**”. The idea is to provide a framework in which it is easy to build specific tools from generic components. We strongly believe that this approach is needed in the domain of software exploration, at least in the context of large companies. This paper presents G^{SEE} , a Generic Software Exploration Environment.

The rest of this paper is structured as follows. Section 2 presents the requirements. Section 3 describes our approach. Section 4 illustrates the approach by presenting some key aspects of the G^{SEE} object-oriented framework. Section 5 illustrates the use of different tools based on the G^{SEE} framework. Section 6 provides a discussion.

2. G^{SEE} Requirements

2.1. Background

The main trend of research on reverse engineering focuses on programming-in-the-small models. By contrast, the requirements for GSEE are based on our experience in building reverse engineering tools to deal with non-traditional models including configuration management models (e.g. [8,7,9]) and component models (e.g. [10,22,26]). In particular, the need for generic tools has been further put in evidence in the context of a collaboration between the LSR laboratory and a large software company, Dassault Systèmes (DS).

DS is the world's leader in CAD/CAM markets, thanks to its main product lines: CATIA, ENOVIA and DELMIA. Our study focused on CATIA, a large software product developed by more than 1000 developers. This software is made of a huge amount of entities (4 M LOC; 50 000 C++ classes; 1,000,000 files for all versions). It relies on a wide range of software models and software representations.

One of the aims of the LSR-DS collaboration was to formalize one of the many software models used at DS: the component model (OM). We built a specific exploration tool, the OM Visualization Tool (OMVT), dedicated to the visualization and exploration of OM components. The ability to visualize existing components was very appreciated since more than 4,000 components had already been built by DS. For the first time OM components were graphically represented in a concise and understandable way. This was an important achievement since many components are made of thousands of lines of code spread over many files.

OMVT tool represents a significant development effort. The success of the first version naturally brought up the

problem of its evolution and generalization. Interviews with DS software engineers revealed the existence of many other perspectives and software models. It also showed us that there are a great diversity of potential users for such exploration tools.

We concluded that, in a such a context, what is needed is a Generic Software Exploration Environment (G^{SEE}) that allows to produce an arbitrary set of views on an arbitrary set of data at almost no cost.

2.2. Requirements

Based on the background described above, we defined a set of requirements for G^{SEE} . Similar requirements have been reported in the literature, sometimes in different ways. This paper concentrates only on a few of these requirements.

Multi-source exploration. The exploration environment should be independent from the source of data. It should be possible to display virtually any kind of structured data. Views could result from the fusion of different heterogeneous data sources [17]. It should also be possible to integrate into the environment a new data source, either programmatically or dynamically, without an extensive knowledge of its structure.

Multi-visualization exploration. It should be possible to represent the same piece of information in many different ways [28]. Swapping from a visualization technique to another should be very easy. It should be possible to customize views interactively [30].

Customizable exploration. Exploration needs and skills greatly vary over a large company. The environment should be able to provide a wide range of powerful features to the experienced explorer, but a small set of simple functions to the novice [30]. The environment should also provide features in order to create specific tools for specific tasks. In this case, it is probably better to use the term inspection rather than exploration. The graphical interface of these specific inspection tools must be customized to fulfill the specific needs in an efficient way [28,30]. For instance, these interfaces can consist in specific views on the screen arranged within a precise layout [18].

3. The G^{SEE} Approach

Today, large successful software companies like DS strongly rely on object-oriented (OO) and component-based (CB) techniques. We believe that the availability of these techniques provide new opportunities for understanding and reverse engineering tools, and in particular for a generic software exploration environment. Our approach consists in providing (1) an object-oriented framework, (2) a set of customizable tools and (3) a tool builder.

3.1. An object-oriented framework

Instead of providing a huge generic tool, we believe that the first step is to provide an object-oriented framework dedicated to software exploration. Simply put, an OO framework is a well organized set of packages with well defined interfaces. The framework approach brings flexibility and extensibility, while making almost no assumption about the way it will be used. It is possible to reuse only a part of the framework, to compose the given elements in many ways, and even to extend the framework when a new technique is available.

3.2. A set of customizable tools

A set of customizable tools must be delivered with the framework. These tools can serve to illustrate the use of the framework. They also enable the user to get immediate functionality without requiring programming effort. Depending on the intended audience, different tools can be proposed, ranging from tools with a user-friendly interface but limited expressive power, to highly customizable environments based on an interpreter of a high-level (query) language. Whatever the interface, the common idea behind these tools is that they should provide some way to be adapted to specific needs.

3.3. A tool builder

Customizable tools are not enough: simple tools may be too limited in scope, while highly generic tools are often too complex to use. In an industrial context, tools used daily by hundreds of software engineers must be designed very carefully. The programmatic effort required to build such specific tools from pieces supplied by the framework may be considered too high. In this case a tool builder offer an good alternative.

The idea here is to provide a tool generator, referred as the “tool builder” making it possible to build a specific exploration tool by interactively assembling components. This approach, based on the component technology, is exemplified by the Java Beans component model designed by Sun [29].

Currently only the object-oriented framework, and some customizable tools have been implemented. We are in the process of designing a new component model on top of Java Beans to get rid of its limitations.

4. The G^{SEE} Framework

The G^{SEE} framework supports a large set of features but, to illustrate our approach, we can focus on two important aspects of any exploration tools: (1) the extraction of information from the software (2) the visualization or presentation of this information. G^{SEE} is thus based on two kind of components: source components and visualization components.

4.1. Source components

Source components are those components responsible for giving access to the software artifacts in an uniform way, irrespective of the actual software representation, software models and even software meta models. To do this we took a solution combining the object-oriented and functional paradigms. This integration has been done in different context including programming languages (e.g. [19]), temporal and versioned data bases (e.g. [31]). Below, we give a rough idea of this approach, focusing on one of its central parts: the integration of the concept of function in the object-oriented world.

As any OO framework, G^{SEE} is based on some interfaces and associated implementations. Indeed, the whole framework is centered around few very simple interfaces. Since those interfaces define the requirements on the data to be explored they should be as simple as possible. Actually, implementing a single interface, the *Successor* interface, is sufficient to get almost all G^{SEE} functionality. This paper concentrates on this interface.

4.1.1. The Successor interface. The *Successor* interface traduces, in the object-oriented model, the notion of function in the mathematical sense of the term [27], or to be more precise the notion of multi-valued function. A multi-valued function is a function yielding for a given object a collection of objects.

```
interface Successor {
    Collection getSuccess(Object o) ;
}
```

The strongest point of this interface is its simplicity: it contains a single method. It is therefore easy to implement, reducing the effort required to add a new data source.

Note that this interface is designed to return a collection rather than an object, because a collection is a versatile data type, representing an abstraction of sets, multi-sets (bags), sequences, etc. Moreover a single object can easily be converted to a singleton collection.

While being simple this interface is also extremely versatile: the concept of multi-valued function is sufficient to navigate between entities, to get their attributes, etc.

For instance, implementing a file browser only require to provide an implementation returning, for a given directory, the set of files and directories it is composed of.

Thanks to abstraction provided by the use of an interface, G^{SEE} is able to deal with any kind of implementation. On the one hand, the implementation of the function may be based on a data structure, representing the function *in extension* [27] (typically a function can be seen as a set of pair values). On the other hand, the function may be represented *in comprehension*, that is to say by means of a piece of code computing on-the-fly, for a given object, the set of its successors. As we will see, this is one of the most interesting feature of our approach.

Thanks to the `Successor` interface, it is possible to explore virtually any source of data; a programmer can provide his own *specific implementation* suited to his specific data source. As point out before, this is especially easy since only one method has to be implemented.

However, the full power of the G^{SEE} framework comes from the availability of a wide range of implementations. The set of implementations provided are of two kinds: (1) *basic generic implementations*: they play the role of wrappers to some piece of data either represented in extension or in comprehension, (2) *compositional implementations* which are implementations combining other implementations (either basic or composite) to build new implementations. As we will see, the composition can be done either programmatically or interactively.

4.1.2. Basic implementations in extension. Generic implementation are wrapping data structures including text files, databases and object attributes.

Wrappers to text files. Using flat ASCII file formats is probably one of the most common ways to represent explicitly software entities and relations. So, we have developed a set of wrappers that build successors from informations stored in such files. The diversity of file formats is handled through the use of different parameterized implementations. For instance `PairsFileSuccessor` is a particular implementation of the `Successor` interface based on a set of string pairs stored in a file. This class provides various constructors and methods allowing to set the value of parameters like field separators, etc. Default values makes it very easy to deal with the most common cases. Similarly some wrappers are provided to support the Rigi Standard File format [21]. We are currently studying how to handle XML files and GXL format [12].

Wrappers to databases. G^{SEE} also support access to relational databases: two columns of a relational table is an explicit representation of a multi-valued function, that is a successor. Interoperability with relational databases is simple thanks to the JDBC interface [29]. This industrial

standard enables the interoperability with virtually any relational database system, even remotely located. G^{SEE} also provides support for object-oriented databases. In particular we have implemented a bridge to the Object Store database system in the context of our collaboration with Dassault Systeme [10]. Currently, interoperability with Object-Oriented database systems requires almost no effort, since the goal of these systems is to make a transparent access to the database. Once connected, exploring this kind of data is just as simple as exploring a graph of objects in memory.

Wrappers to attributes. Dealing with arbitrary structure of objects represented in memory, is one of the most powerful features of G^{SEE} . While most reverse engineering environments require information extracted from the software to be stored in a persistent way, G^{SEE} is able to get the information directly from the tool that generates it (if it can be connected to the G^{SEE} process in some way). For instance it is possible to get information from the abstract syntax tree directly generated by a parser (as long as this data structure can be accessed by the G^{SEE} process). These features can be realized directly by means of implementations provided by G^{SEE} . Some implementations, like `MapSuccessor`, wrap associative data structures like hashtables into successors. Each attribute of a class can be viewed as a successor yielding, for a given object, the value of the specified attribute. A single implementation called `NamedFieldSuccessor` is enough for that. This implementation takes the name of the attribute as parameter and later uses introspection to access to the attribute. Thanks to this feature it is possible to access to arbitrary set of objects without writing any piece of code.

4.1.3. Basic implementations in comprehension. G^{SEE} do not require the information about the software to be represented explicitly: some basic implementation handle data in comprehension (that is to say by means of a piece of code). This includes for example wrappers to methods, shell scripts, and database queries.

Wrappers to object methods. Object methods are natural candidates to implement the `Successor` interface. This is possible thanks to `NamedMethodSuccessor`. For instance, the next expression constructs the unique successor required to build a file browser (`listFiles` is the name of the java method returning a directory's content).

```
new NamedMethodSuccessor("listFiles")
```

Wrappers to batch commands. In large companies, software repositories are often made available only through batch commands and shell scripts. This is typically the case of configuration management systems. It is therefore

important to provide wrappers able to exploit this kind of information. For instance, `ShellCommandSuccessor` takes as parameter a string describing a command to be executed. Different other parameters indicate how to parse the resulting output.

Wrappers to database queries. As said above information stored in a database can be directly accessed. Derived information (represented in comprehension) can also be obtained thanks to the execution of queries. A query, for instance an SQL query, is thus a potential implementation of the successor interface.

4.1.4. Compositional implementations. The basic implementations presented above give access to existing data (either represented explicitly or not). In many cases, it is also necessary to derive new information from this data. This is what the compositional implementations are for: they enable to create complex successors by connecting elementary successors with well-defined operators. G^{SEE} provides a large set of such operators to cover a wide range of needs.

The first set of operators provided by G^{SEE} has been directly derived from the mathematical toolkit of the Z formal specification language [27] and is therefore based on the set theory.

For instance the `UnionSuccessor` implementation takes two successors as parameters. For a given object it yields the union of the results returned by those successors. Similarly `ComposeSuccessor` returns the composition of two successors: the second one being applied to the result of the first one. More interestingly `ClosureSuccessor` computes the transitive closure of a given successor, that is to say for a given object it returns the set of all objects directly or indirectly accessible. `SelectSuccessor` takes a predicate and a successor and return only those objects satisfying the predicate, etc. G^{SEE} also include aggregate functions like cardinality, sum, etc.

Some compositional implementations are also provided to deal with non-functional requirements. Even if these operators have no effect on the results yield by the underlying successor, they allow for instance trade-offs between execution speed and memory requirement. This is the case for the `BufferedSuccessor` operator taking a successor as argument and memorizing on-the-fly the last calls to the function to avoid unnecessary recomputation (this technique is known as “memoization” in functional languages). In particular this operator can be used to convert a function described in comprehension to a function represented in extension, improving greatly performances for computing intensive function. Optionally, the result can be made persistent.

4.1.5. Programmatic vs. interactive composition.

Since the compositional implementations described above are based on successors and implement themselves the `Successor` interface, they can be recursively composed to build sophisticated functions. In particular it is possible to merge different sources of data in a transparent way, for instance data coming from a relational database with the transitive closure of data obtained from plain files and composed with functions implemented by a batch system and buffered to avoid unnecessary computation. Moreover, the java language provides a convenient notation for inline classes enabling the programmer to easily include java code within arbitrary successor expressions. This is exemplified by the next piece of code.

```
new UnionSuccessor (
  new ClosureSuccessor (
    new BufferedSuccessor (
      new Successor() {
        public Collection getSuccs(Object o) {
          // here is some piece of java code
        }
      }
    ),
    new NamedMethodSuccessor("listfiles") )
```

Even if the code above seems at the first sight complicated, it is much simpler than if it were implemented from scratch. Moreover it is certainly safer since the latter solution would require to implement the transitive closure and buffering algorithms by hand. This is error prone.

To further ease the composition of successor implementations, G^{SEE} also provides a textual language giving a concise and precise notation for the different operators. For instance in the example below, the transitive closure (*) of the method `listfiles` is composed ($;$) with the `getName` method: this successor returns for a given directory the name of all files it contains directly or indirectly. The notation for these operators is directly inspired from the Z notation [27].

```
new GSLSuccessor("listfiles^*;getName")
```

Even if this feature is useful in the programmatic approach, its main benefit comes from the ability to integrate easily this textual language within interactive exploration tools. This point will be illustrated in section 5. Indeed this language can be seen as a functional query language. While the current version of G^{SEE} only supports textual languages, we are currently extending this environment to support a graphical syntax similar in spirit to the Hy+ query language [20].

Finally note that implicit type conversion takes places within these expressions whenever required to obtain a well typed expression. This helps using such expressions in an interactive way. It also makes the language very concise.

4.2. Visualization components

While the interfaces and implementations described above deal with the sources of data, visualization constitutes another important part of the G^{SEE} framework. Indeed the interfaces required by visualization components are also expressed in terms of abstract structures based on the set theory: sequence, graph, tree, etc. In other words all data displayed by visualization is expressed in terms of type constructors that can in turn be expressed in terms of set and functions. This uniform treatment of components greatly helps the connection between source components and visualization components.

From a concrete point of view, G^{SEE} includes a large set of visualization components. G^{SEE} is based on the Java Bean component model [29] and makes an extensive use of the Swing framework provided with the java environment. In particular this framework provided valuable components to visualize a rich set of structures including sequences, tables, trees, hyper texts, etc. To complete the spectrum of visualization techniques, G^{SEE} also integrates wrappers to various other visualization components, such as Grappa, the java version of the dot graph visualization tool [11]. We also have developed from scratch different visualization components such as tree maps and line sequences inspired from [6]. All visualization techniques currently available in G^{SEE} have been selected for their ability to display very large sets of data.

Tough a wide range of components are included with G^{SEE} , it is still possible to add to the environment new visualization components dynamically, just like source components. This makes it possible to include specific components.

An interesting aspect of G^{SEE} , is that each visualization component described above has been encapsulated to support a uniform interface. All views are described in the same format: (1) a model specifies the software artifacts to visualize, and (2) a renderer indicates how these artifacts are mapped to graphical entities. This approach followed by most modern visualization frameworks, is further improved in G^{SEE} : both the model and the renderer can be expressed in terms of one or more successors, making it very easy to produce a new view.

To further simplify the production of renderers, the G^{SEE} framework also provides a set of interfaces and implementations dedicated to visualization. For instance the interface `Colorizer` is intended to map objects to colors. The `EnumColorizer` implementation maps a specified set of values to a specified set of colors (this is an example of function represented in extension since it is a set of pair (value, color)). Similarly `RangeColorizer` maps a range of values to a gradation of color, etc. Other implementations make it possible to combine these features. Some

implementations are provided to edit the renderer properties interactively through the use of panels, color choosers, etc. and to save these renderers for further use.

Since the renderers of each specific component are defined consistently it is easy to switch from a visualization technique to another. For instance, a hierarchical structure can be visualized using a Swing JTree, a graph displayed by Grappa, or a tree map, by just changing a parameter while keeping the same model.

5. The G^{SEE} Customizable Tools

Usually one of the best ways to evaluate a software exploration tool is to see it at work. In the case of G^{SEE} , it is important to keep in mind that the power of this environment is not directly visible since it resides first of all in the G^{SEE} framework. However, in this section two demonstration tools included in the G^{SEE} environment are briefly presented as an illustration of the approach: the G^{SEE} Interpreter and the G^{SEE} Viewer. Though simple, these tools proved to be usable on a very large scale, in the context of Dassault Syst me [10,26], on a software made of more than 40 000 C++ classes. In the context of this paper, let us suppose that the goal is to explore the java standard library (more than 8000 java classes are delivered with the JDK1.3). The same tools can be used without any modification: instead of loading DS' repository at the beginning of the session a connection will be made to source components extracting information from java programs.

5.1. Example 1: the G^{SEE} Interpreter

The first demonstration tool is called the G^{SEE} Interpreter (see Figure 1). Thanks to the framework, this tool is made of only 60 lines of java code!

5.1.1. Features. This tool aims to give access to the G^{SEE} language through a very rudimentary interface (see Figure 1). Simply put, the G^{SEE} language is a functional language giving access to the compositional operators supplied by the framework.

Each step of interaction with the G^{SEE} Interpreter consists in entering a new command. The command is immediately interpreted and the result displayed both in a textual and graphical form. In the Figure 1, the history of the session is displayed in textual form on the top. The result of the last command is displayed graphically on the bottom.

There are basically three kinds of commands: (1) expressions (or queries), (2) definitions of new symbols (`let x = ...`), (3) and directives controlling the behavior of the interpreter (e.g. loading a new source component). Like with any other interpreted language, these commands can

be saved to form a program for later use. In particular it is possible to create a specific tool from that program and make this tool available at large for novice explorers.

5.1.2. Example of a scenario. To illustrate the use of the G^{SEE} Interpreter, let us assume that we want to study the relationship between the composition of packages and the inheritance relationship: for instance we want to know, for a given package P, which packages contain the super classes of the classes in P. In other words, we want to know if the inheritance relationship crosses package boundaries. We unlikely want to build from scratch a specific tool for that!

Selecting adequate source components. So let us study the problem and see what we have at hand. Since building a java parser will be far too expensive let see which source components are available and how to extract the necessary information. Thanks to the java introspection library provided with the java environment, it is possible, for a given class to get its super class through the method `getSuperclass`. Unfortunately, this library does not provide enough support to deal with packages. For instance it is not possible for a given package to get the list of the classes it contains. Before starting to write a tedious piece of code, it is a good idea to check if there is on Internet some piece of code already providing this functionality. After a search among the many tools freely available (e.g. [1,3,15]), JavaAssistant is found [4]. The main page describes shortly the functionality of the tool. In this page the following sentence is found “*adavid.reflect.PackageFinder: finds all the packages on your system*”. To get further information the tool is downloaded.

Loading selected source components under G^{SEE} . The JavaAssistant is a specific tool with a specific set of features. However, from the G^{SEE} point of view JavaAssistant could be considered as a source component since it provides some way to extract information from java programs. So let’s start a session with the G^{SEE} Interpreter, and load this java component (see step (1) in Figure 1).

Exploring source components. Since we have just loaded a new component, we do not know much about it. Fortunately, we can use G^{SEE} to explore the software model implemented by this component: after all this is just another piece of software. Actually G^{SEE} supports the exploration of software models, that is, the exploration of meta information on software. This topic is an important feature of G^{SEE} but is out the scope of this paper. This step has not been shown in the figure so the whole scenario can fit into the history window. What is important here, is that we learn that the method `getPackageResources` gives access to packages.

Building new functions and getting the result. Before all, a short name is given for that method (2). The function is then tested on a package, for example

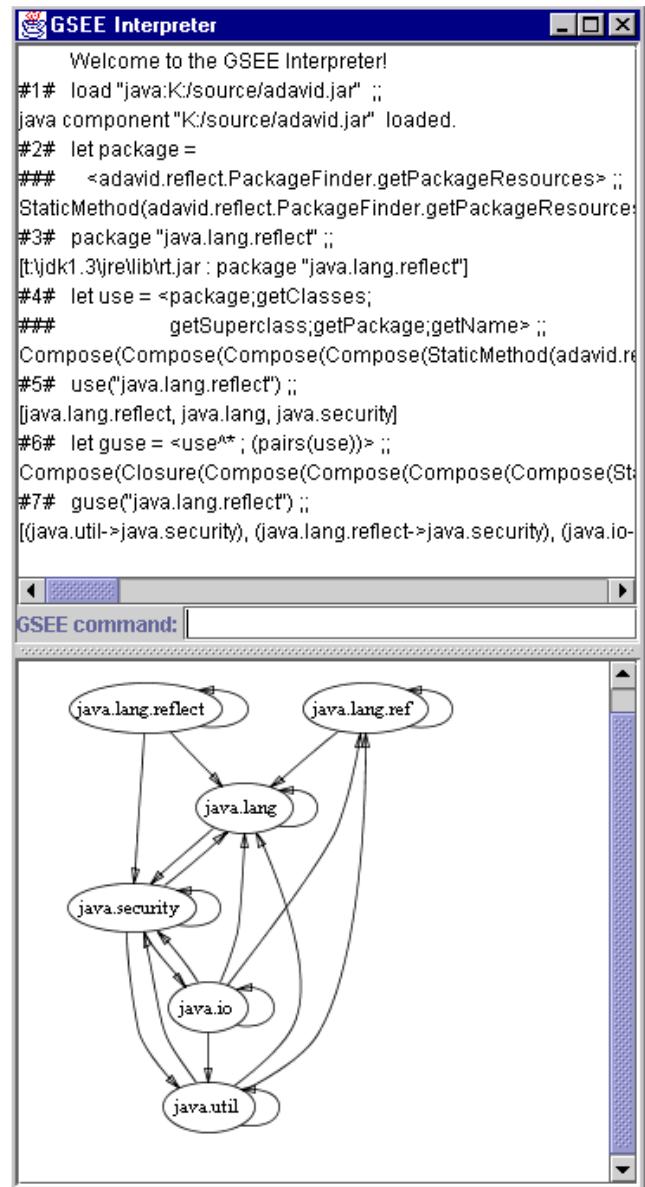


Figure 1. A session with the G^{SEE} Interpreter

`java.lang.reflect` (3). It seems to work, so we now define a successor use as being the function we need (4). The successor expression

```
package;getClasses;getSuperclass
;getPackage;getName
```

means that we want “the names of the packages that contain the super classes of the classes contained in a given package”. The main benefit of an interpreted language is that we can try it immediately (5). From the output, we learn that the `java.lang.reflect` package “uses” three packages, namely `java.lang`, `java.security`, and

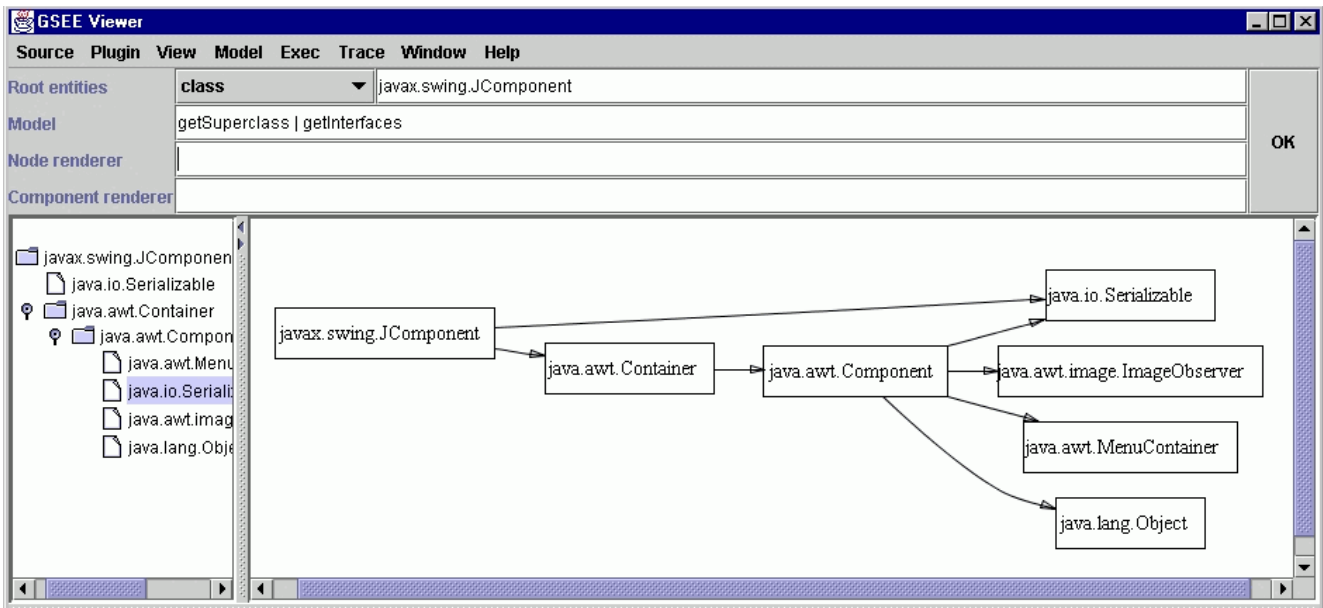


Figure 2. A simple view built with the G^{SEE} Viewer : superclasses and interfaces of the class JComponent

`java.lang.reflect`. To get a global view, a function returning a graph is defined (6) and tested (7). That's all we want. We have got the graph on the bottom of the window. As we can see, the core packages of java are strongly connected!

Creating a new specific tool. We have just defined interactively a new function taking as parameter one or more package names and displaying the graph of inheritance over these packages. This function is useful so we may want to save the program we have just built and made it available to the whole team. Since other software engineers do not know much about the G^{SEE} language, we supply them a specific tool with a simplified interface: a text field to enter the name of package and a panel to display the graph. This tool can be standalone or integrated as a plug-in in a programming environment. Currently G^{SEE} is able to create plug-ins for the Kawa programming environment [16]. The function can be called directly from Kawa menus. The integration is therefore entirely transparent to the novice explorer. He can use its favorite programming environment without even knowing that this is actually a G^{SEE} plug-in.

5.2. Example 2: the G^{SEE} Viewer

Instead of using a complex language to build complex expressions, the G^{SEE} Viewer is based on panels, buttons and menus to guide the explorer. The main idea behind this tool is that views can be specified at a high level of abstraction by a wide range of explorers, as long as they know what information is required.

5.2.1. Features. The G^{SEE} Viewer is also a demonstration tool. Tough its current interface can be considerably improved, this tool has proved to be very useful in practice [10,26].

As illustrated in Figures 2 and 4, views are described by different fields specifying (1) the information to display (the model) and (2) how to display it (the renderer). To be more precise, the first line of the specification specifies the root entities. The next line indicates the relationship(s) to be displayed. Finally the last two lines specify the rendering, by providing the value of different properties such as the color of the nodes, the space between the nodes, etc. Figure 4 includes an example of a rendering specification.

Figures 2 and Figure 4 depict a graphical user interface in which the edition of the rendering specification is done by textual editing of the last two text fields. The configuration of the rendering could have been done through the use of a more sophisticated graphical interface made of panels, buttons, slide bars, etc. Indeed, just like in the Java Bean component model, the edition of a component configuration is a responsibility that can be shared by the environment and the component itself [29]. In other words, a visualization component can provide for its configuration editor its own piece of code. If it does not, a default editor will be used. As shown in Figure 4 in that case the properties are specified by means of a sequence of a string pair values (e.g. `color = ... width = ...`).

Usually each property of a Java Bean component is set once and has a constant value. With this approach it is possible for instance to indicate that the background color of the component is yellow. The approach retained by G^{SEE}

is much more powerful: a property can be set either to a constant or to a function. This makes it possible for instance to set the color of each node to an information related to this node. The functions are directly expressed in terms of the model making thus the connection between the visualization components and the source components. From a concrete point of view, the different expressions filling the various fields are successor expressions of arbitrary complexity. Expert explorer can use sophisticated expressions, while novice explorers may use simple ones.

One of the major benefit of an interactive system like the G^{SEE} Viewer is that the explorer can build the view specification incrementally, getting at each step the associated result. In particular he or she can start for instance with blank lines (see Figure 2): default values enable to get immediately a first result. Then it is possible to make successive adjustments by adding/changing properties and pressing the OK button to update the view(s). When considered satisfactory, views can be named and saved, so that novice explorers can later load available views to explore their own software without any knowledge of the G^{SEE} language.

5.2.2. Example of a session. To illustrate the use of the G^{SEE} Interpreter let's continue the scenario started in the previous section: somebody in the company liked the ability to visualize the inheritance at the level of packages but now he wants to get a more precise picture of it at the level of classes.

Starting the G^{SEE} Viewer. He starts the G^{SEE} Viewer with an initial environment including the functions we have already defined in Section 5.1. Those functions are made available when the tool starts so the tool behavior is just like as if it were specialized for the exploration of java programs. In particular the menu on the first line contains items like `class`, `classes of package`, etc. Nothing indicates that different heterogeneous source components cooperate behind the scenes to give the results.

Building a simple view (Figure 2, 3a). To start, the explorer just wants to see a hierarchy of classes and interfaces for a given class. He first selects the `class` item in the menu, enters the name of the class, for instance `javax.swing.JComponent`, and press OK. The entities

appears on the screen. Setting the model line to `getSuperclass | getInterfaces` now displays the graph depicted on the right of Figure 2 (he have just learned the name of these methods thanks to a contextual menu displayed over the class just displayed). Though the rendering lines have been left in blank, the default values have been applied and the result is satisfactory. Note that by default, a tree has also been displayed on the left of Figure 2. This alternate view displays exactly the same information but in a different format. The explorer can thus choose the best way to interact with the data, depending on the task. Similarly a table is also generated automatically (not shown in the figures).

Defining new metrics and visualizing them (Figure 3b). At that moment a colleague enters the room with the proceedings of WCRE'99 in her hand. She has just read the paper "*An Hybrid Reverse Engineering Approach Combining Metrics and Program Visualization*" by Demeyer and al. [5]. The proposed techniques seem very appealing so she wants to see their effectiveness. They decide to work together to improve the existing view by adding metrics to it. The problem is how to get the metrics and how to display them. From a menu they learn that one of the visualization components currently loaded (`grappa`) supports a `width` and `height` properties for each node. With respect to the availability of metrics, no metrics are directly provided by the available source components. It does not matter. G^{SEE} can be used to derive new metrics from existing data and compute them on the fly, without the need to change the software models just loaded. It is possible to get the list of methods and fields declared with a given class. Counting these elements gives a simple yet interesting metrics [5]. So they decide to set the rendering specification with the following properties:

```
width=getDeclaredFields#
height=getDeclaredMethods#
```

Pressing the OK button gives a new view on the same information but with a very different perspective: a strong emphasis being put on "big" classes. As Figure 3b shows the classes `JComponent` and `Component` are far more complex than the other ones.

Mapping package to colors (Figure 3c). As suggested in [5] the color could be used to map a third metric. Thanks to the colorizers described in Section 4.2, it is easy to map a

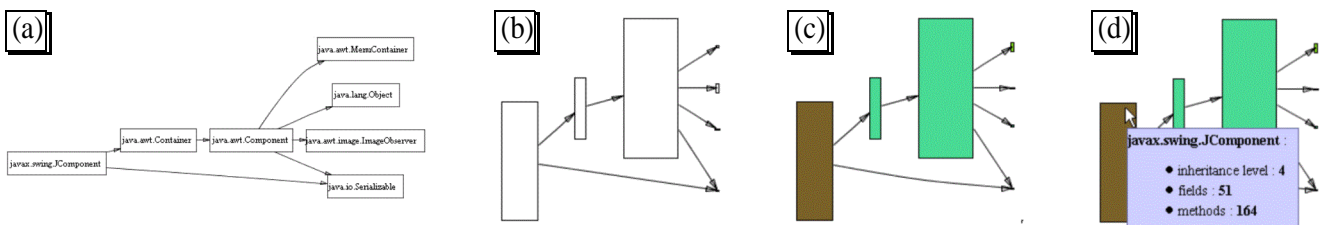


Figure 3. Incremental design of a view on software

range of values to a color. In the case of the current scenario, the explorers prefer to study the relationship between the inheritance and the composition of packages (see Section 5.1) so they decide to map the color of each class to the package containing that class. Interestingly, adding the property `color=getPackage` is enough to get a first result (Figure 3c). Obviously the function `getPackage` returns a package, not a color. But this is not an issue because as said before, G^{SEE} is able to handle type mismatches and inserts type conversion operators when necessary. In that case G^{SEE} builds automatically a new colorizer mapping each distinct values to a random color, ensuring that all classes of a given package are represented with the same color. The ability to create automatically a first colorizer is an important feature in practice because (1) it is not possible to know a priori what packages are involved in an inheritance graph, (2) many packages may be involved (3) the explorers may not have particular preferences about the color coding. Though the packages are randomly mapped to color, if they feel the need for it, the explorers may look at a caption that is automatically generated. What is more, since colorizers are themselves components, the explorers can change them interactively, save them, load existing ones, etc. This helps maintaining consistency between views over the company. In the case of the scenario, the explorers don't change the colorizer: they just want to see the change of color along inheritance path.

Adding further information to the view (Figure 3d). The view currently displayed is interesting because it gives an global view. However it would be very interesting if it was possible to get more information on a particular class by just moving the cursor on it. The `tip` attribute of the visualization component can be used for that. After adding the property `tip=getName` to display the name of the class the explorers decide to provide more information. The HTML format is used to improved the readability of the message displayed. Pressing OK and moving the cursor to a class makes it possible to test immediately the effect of the different HTML markers. Thanks to this technique, the explorers learn that the `JComponent` class is really a big one: it declares 164 methods! By just moving the cursor to the `Component` class on the left, they learn this class declares 195 methods and 76 attributes...

Getting a global view on a large piece of software (Figure 4). The result is satisfactory, so it is time to see if the view just designed, scales up. The explorers thus decide to look at the set of classes included in the packages `java.awt` and `javax.swing` (these packages are those used to build graphical interfaces with java language). Instead of starting from one class, the graph traversal has to start from all classes contained in these packages. To get a clear picture, only class inheritance is displayed. After changing accordingly the model specification and few

rendering options, an interesting picture appears on the screen (computing this view takes 6 seconds on a PC with a 700Mhz processor). Figure 4 displays the inheritance relationships between 689 classes!

The difference in size between classes is striking. There are only a dozen big classes, the remaining 600 are really small! Interestingly all "big" classes are roughly in the same hierarchy. Brown classes (a) are on the bottom of this hierarchy, while green ones are on the top (b). By moving the cursor on the big classes, the explorers rapidly recognize the graph they have just studied previously: `Component` (a) and `JComponent` (b) are the biggest classes. The picture clearly shows that the swing toolkit is based on the AWT toolkit. This can confirm the knowledge an experienced java programmer could have.

A closer examination reveals many interesting points. For instance disconnected nodes (c) are java interfaces (they are not linked by the super class inheritance): there are very few interfaces but many classes. The usage of the interface notion is rather low within this framework.

Attention of the explorers is then drawn to the node marked (d) because of its very peculiar shape: it has many fields (194!), but only few methods. This class should be considered suspect at the first sight. Moving the cursor on it reveals its name: `java.awt.event.KeyEvent`. This is thus a false alarm: this class deals with keystrokes and is made of many definition of constant fields.

In the same region it is interesting to note that the color associated to that class constitutes a whole sub-tree (all nodes in the sub-tree (f) are blue). This indicates a good cohesion of the package since the variants of a same concept are grouped within a single package (actually these class are events, the package name is `java.awt.event`).

At that moment, another colleague enter the room. Impressed by the view displayed on the screen, he wants to slightly change it to display the age and author of each class. He is responsible of the configuration management team and knows how to extract this information from some batch commands. So the session continues...

6. Discussion

Various remarks should be made about the scenario described above.

G^{SEE} is not an exploration tool. The G^{SEE} Viewer and Interpreter are very simple demonstration tools with simple interface. Actually, these tools are only the tip of the iceberg. The full power of the G^{SEE} lies in the flexibility of its object-oriented framework. In particular, it is possible to reuse only few classes of the framework to create an application. G^{SEE} can also create a plug-in that will be integrated in an existing programming environment. This approach contrasts with most other exploration tools.

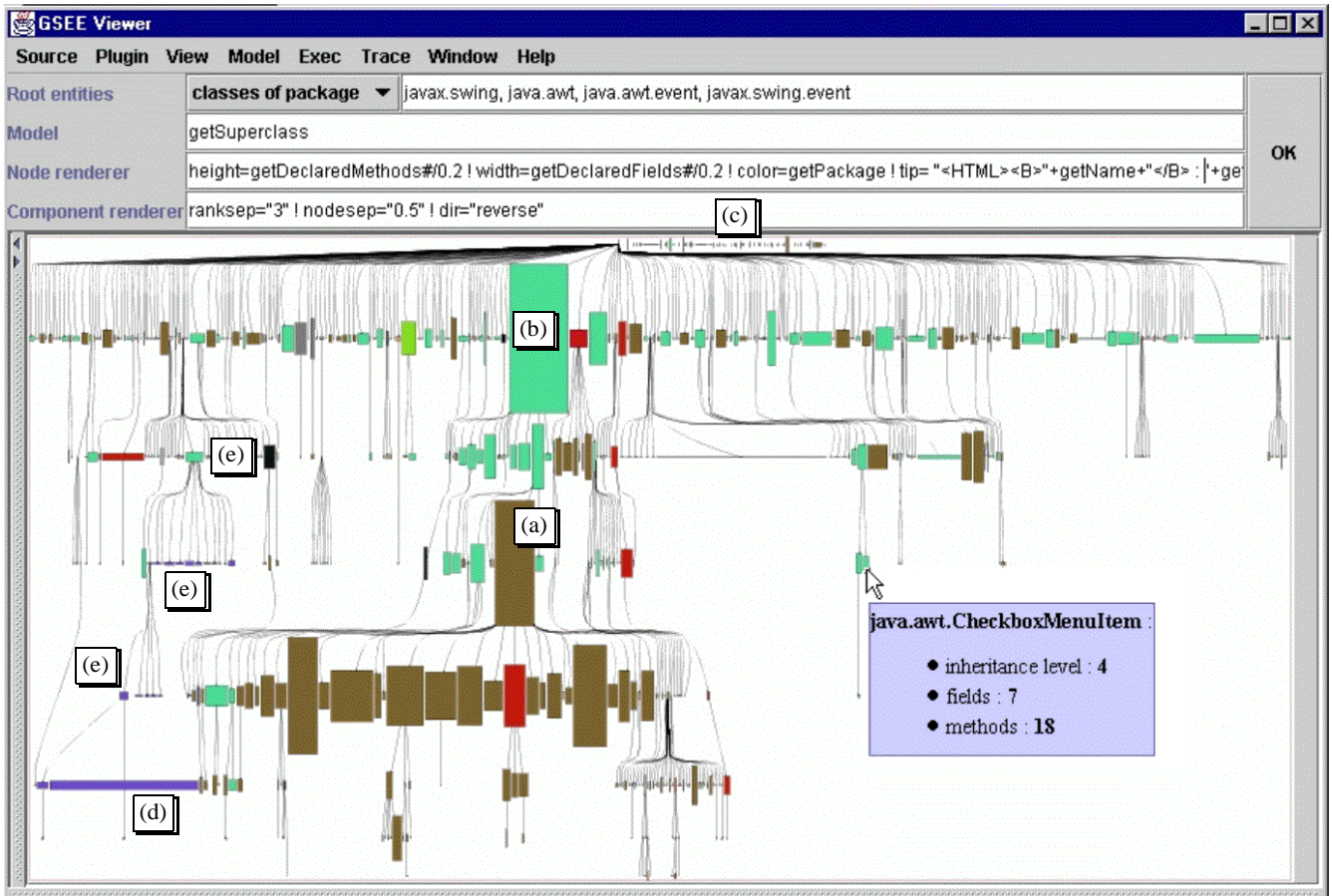


Figure 4. Inheritance relationship

G^{SEE} is not linked to a particular software model. G^{SEE} is not restricted in any way to the exploration of java programs. G^{SEE} has been applied on many different other sources of data including for instance file structures, software processes or component models, etc. Indeed, G^{SEE} could be used to explore any piece of data, not only software.

G^{SEE} is not linked to a particular representation. The scenario described above is based on the integration of a piece of code extracting information about the software. However, as explained in Section 4.1, G^{SEE} is also able to access information from flat files, databases, batch systems, etc. The G^{SEE} philosophy consists in looking for the data where it is, without unnecessary data conversion. In fact, the ability to deal with representation in comprehension is actually one of the strongest features of G^{SEE} . By contrast, most existing exploration tools are based on data previously extracted and represented in extension by means of a file exchange format such as RSF [21] or GXL [12]. This approach has some limitations: “the size of exchanged GXL files will be somewhat large”; “API to access information about a program is sometimes considerably more efficient and convenient than a stream such as GXL” [12]. G^{SEE}

smoothly integrates both approaches allowing thus to explore on demand huge software structures.

G^{SEE} is not linked to a particular visualization. Most exploration tools are based on a particular visualization technique. For instance the Rigi environment is centered around graph visualization. G^{SEE} is not bound to a specific technique. New visualization components can be loaded dynamically.

7. Conclusion

Large software companies manage large software products. They not only have to deal with many software entities, but also with many software models and many software representations. Tools could help in the understanding of this huge amount of information, but the problem is thus to be able to build such tools and then to make them evolve. In this paper we claim that a generic approach should be taken in such a context.

G^{SEE} is a Generic Software Exploration Environment. It makes it possible to specify and create new exploration tools very easily. G^{SEE} is really generic: it does not depend on a particular kind of data nor on a particular visualization

technique. As shown in this paper, only a few lines are necessary to build new views and get interesting results.

From a concrete point of view, G^{SEE} is made of an object-oriented framework and a set of customizable tools built on the top of it. Thanks to a small set of interfaces, but a wide range of implementations, the G^{SEE} framework is simple but powerful. For instance, the G^{SEE} Interpreter is made of only 60 lines of java code. The framework will naturally continue to evolve as well as the set of associated tools.

An interesting aspect of G^{SEE} , is that new source components can be connected on the fly, during an exploration session. In particular, the explorer may continue the exploration via an unknown source components. In that case, the problem is not only to explore software entities but to explore the software model itself. We believe that this last point constitutes an important research issue for the future. G^{SEE} already provides first solutions to address this problem. This will be the topic of another paper.

Currently we are in the process of defining a new component model on top of Java Beans. This component model will form the basis of the G^{SEE} tool builder, and will be used to develop further reverse engineering components. In fact, we strongly believe that merging reverse engineering and component-based software engineering is a promising approach. We have already made first steps in that direction: on the one hand in [10] G^{SEE} is applied to the reverse engineering of component-based software; on the other hand this paper describes G^{SEE} , a reverse engineering software based on components.

More information about G^{SEE} can be found at the following URL: <http://www-adele.imag.fr/~jmfavre/GSEE>.

8. References

- [1] B. Bokowski, A. Spiegel, "Barat - a front-end for Java", Technical Report, TR-B-98-09, Univ. Berlin, Dec. 1998. <http://www.inf.fu-berlin.de/~bokowski>
- [2] CIA/++,CIAO <http://www.research.att.com/~ciao/>
- [3] M. Dahm, "Byte Code Engineering with the JavaClass API", Tech. Report, TR-B-17-98, Univ. Berlin, Dec. 1998.
- [4] A. David, "JavaAssistant 1.6, On-the-fly Class Browser", <http://www.docs.uu.se/~adavid>
- [5] 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.
- [6] 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.
- [7] 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
- [8] J.M. Favre, "Understanding-In-The-Large", Proc. of the 5th International Workshop on Program Comprehension (IWPC'97), IEEE, May 1997.
- [9] 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.
- [10] 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.
- [11] Graphviz, <http://www.graphviz.org/>
- [12] R.C. Holt, A. Winter, A. Schür, "GXL: Toward a Standard Exchange Format", Tech. Report of Univ. Koblenz-Landau, May 2000.
- [13] R. Holt et al, PBS: Portable Bookshelf Tools, <http://www.turing.toronto.edu>
- [14] Imagix, <http://www.imagix.com>
- [15] JavaSrc, <http://home.austin.rr.com/kjohnston/javasrc.htm>
- [16] Kawa IDE. <http://www.tek-tools.com/kawa/>
- [17] R. Kazman, S.J. Carriere, "View extraction and view fusion in architectural understanding", Proc. of the 5th Int. Conference on Software Reuse, 1998.
- [18] 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.
- [19] M.V. Mannino, I.J. Choi, D.S. Batory, "The Object-Oriented Functional Data Language", IEEE Transactions On Software Engineering, Vol. 16, No. 11, Nov. 1990.
- [20] A. Mendelzon, J. Sametinger, "Reverse Engineering by Visualizing and Querying"
- [21] H.A. Muller et al, RIGI, <http://www.rigi.csc.uvic.ca/>
- [22] S.T. Nguyen, J.M. Favre, Y. Ledru, J. Estublier, "Exploring Large Software Products", Proc. of ICSSEA, Paris, Dec 2000 (in french).
- [23] D.E. Perry, "Software Interconnection Models", Proc. of the 9th Int. Conf. On Software Engineering, IEEE, March 1987.
- [24] S. Robitaille, R. Schauer, and R.K. Keller, "Bridging Program Comprehension Tools by Design Navigation", Proc. of the Intl. Conf. on Software Maintenance, Oct. 2000.
- [25] S. Tichelaar, M. Lanza, S. Ducasse, "Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems", Proc. of the 2nd Int. Symp. on Constructing Software Engineering Tools, June 2000.
- [26] R. Sanlaville, J.M. Favre, Y. Ledru, "Helping Various Stakeholders to Understand a Very Large Software Product" submitted to the European Conference on Component-Based Software Engineering, 2001
- [27] Spivey, "The Z Notation", Prentice Hall
- [28] M.A. Storey, F.D. Fracchia, H.A. Müller, "Cognitive design elements to support the construction of a mental model during software exploration", Proc. of the 5th International Workshop on Program Comprehension, IEEE, May 1997.
- [29] Sun, <http://java.sun.com>
- [30] S.R. Tilley, S. Paul, D.B. Smith, "Towards a Framework for Program Understanding", Proc. of the 4th International Workshop on Program Comprehension (IWPC'96), 1996.
- [31] G.T.J. Wu, U. Dayal; "A Uniform Model for Temporal and Versioned Object-oriented Databases", Chapter 10, in A.U. Tansel, and al. Editors ; "Temporal Databases : Theory, Design, and Implementation", The Benjamin/Cummings Publishing Company, 1993, 635 pages.