

A New Approach to Software Exploration: Back-packing with G^{SEE}

Jean-Marie Favre

*Adele Team, Laboratoire LSR-IMAG
University of Grenoble, France
<http://www-adele.imag.fr/~jmfavre>*

Abstract

Maintaining software is very difficult, not only because of the number of software artifacts, but also because of the large variety of artifacts. Software includes programs, but also makefiles, architectural information, change logs, etc. Different software exploration tools have been proposed in the last decade. Unfortunately, these tools are either specific (e.g. a class browser), or customizable (e.g. Rigi or PBS) but require significant preparation before exploration. This paper presents a new approach to software exploration: software back-packing. This approach allows a simultaneous exploration of software entities and meta-information. Our experience with this approach is briefly presented through G^{SEE}, a Generic Software Exploration Environment.

1. Introduction

While, programming-in-the-small was the first focus of attention in computer science, it is now widely recognized that basic constructs such as statements and procedures are not enough to develop large software products. As a result, new techniques have been provided. Every company has to deal with programming-in-the-large artifacts including configuration scripts, makefiles, change logs, architectural artifacts, test cases, etc. With the shift from traditional programming to component-based development, internet based application development, etc., the variety of entity types involved in a large scale software project is ever increasing. Moreover, large software companies tend to build their own tools suited to their own culture and requirements.

The software world is quite complicated, not only because there are many software entities over the planet, but also because they are many types of entities, some being defined and used only in specific companies.

To help developers and maintainers cope with the complexity of the software they develop, "software

exploration" tools have been proposed (e.g. [1-18]). The corresponding field of research takes its root in "software visualisation" [2] and adds interactivity. Software exploration refers to a process where the user of the tool, becomes a "software explorer" and travels among software entities, discovering new "software landscapes".

This paper shows that this metaphor with real life has not been pushed to its limits. What is true, with current software exploration tools is that you can discover new software entities and new arrangements between these entities, but you will not really discover new things of a new type, new kinds of plants, new animals, or even new creatures. *Specific exploration tools* (e.g. [17][18]) will allow you to travel conveniently but only in restricted areas within specific software countries. *Customizable exploration tools* (e.g. Rigi [13], PBS [11] or GUPRO [12]) require a preparation phase before exploration. The cost of preparation can discourage the potential explorer. As a result, many software countries will never be explored. In this paper, we claim that the "back-packer" approach is worth studying: landing only with a lightweight backpack full of small yet powerful tools, trying to establish direct communication with locals, using their tools, etc.

G^{SEE}, the Generic Software Exploration Environment [20], is a possible solution for software back-packing. At least, this is the one we have constituted along our software travels. It is not very well packed and needs some adjustments, but it has proved to be useful. With it, we go one step further in exploring what software exploration could mean in practice.

The rest of the paper is structured as following. Section 2 describes the background of this research. Then, section 3 reviews three approaches to world exploration: the back-packer, auto-tours and limousines. Section 4 then describes how this metaphor can be used in the context of software exploration. It shows the limitations of current approaches, and introduces *software back-packing*. Section 5 shows how G^{SEE} effectively supports software back-packing. Finally section 6 concludes this paper.

2. Background

Before going further, it is worth situating the context of this research. The back-packer approach does not only emerge from our experience walking in the Alps and in the corridors of the University of Grenoble. It also draws on our experience in trying to explore large industrial software.

This work has one of its roots in the context of a collaboration started in 1996 between the LSR-IMAG academic institution and Dassault Systèmes, one of the largest software companies in Europe. Dassault Systèmes (DS) is the world leader in CAD/CAM. More than 1000 developers work simultaneously on DS' main software products: CATIA and ENOVIA. DS itself represents almost a whole software country, with its own culture and plenty of useful dialects and proprietary tools. In particular DS was, with Microsoft, one of the first software companies in the world to build its own component technology and use it extensively for years [25]. DS has already built more than 8000 components and 50000 C++ classes. To deal with the tremendous requirements related with concurrent engineering, DS developed a sophisticated configuration management system based on the ADELE environment developed at the University of Grenoble [28]. DS maintains a large number of configurations with more than 1 million files. DS has also developed a packaging model to describe the large set of applications sold all over the world. The DS software country contains a huge amount of software artifacts. *The complexity is not only due to the incredibly high numbers of entities, but also to the number of entity types. What is more, these types both evolve over time and space.* Over time to ensure continuous improvement; over space because the different teams in this software country do not have exactly the same needs. As a result, the DS software country could be seen as a fascinating software land with plenty of software structures to explore. Unfortunately, we found that traditional approaches to software exploration failed in a such a specific, evolving and complex context. This leads us to practice software back-packing as described in this paper.

The back-packing approach also results from our previous work on exploring and analyzing non-traditional software artifacts (e.g. [21][22]). Currently, we are also back-packing in the Java world [27]. This software land is characterized by the fact that (1) many artifacts can be visited, and (2) many analysis tools are available (e.g. [33][34][35]). We found that these conditions also constitute a nice opportunity to practice back-packing. We strongly believe that trying to explore different kinds of software country gives us the necessary inputs to ensure the generality of our approach. *These experiences forced us to strongly revise our ideas on software exploration.* In this spirit, let's first study how people travel all over the world.

3. Three approaches to world exploration

They are plenty of ways to travel around the world, but in the context of this paper we can contrast at least three of them: the back-packer approach, the auto-tour, and finally the limousine.

3.1. The back-packer

Let us imagine that you're an old back-packer and that you've decided to explore an foreign country at the furthestmost bounds of the world. Get your back-pack and take a plane or a boat. After landing *exploration can start immediately.* To be successful this approach requires however some empirical skills gained along previous journeys.

Let us review, what could be found in your back-pack. It should be full of *small yet powerful tools.* This could include for instance a Swiss knife, always useful; sun glasses and binoculars to improve vision; a tape recorder, a camera and a notebook to record the maximum amount of information all along your travel, etc.

As a back-packer, during your travel you will try to *communicate directly with locals*, though you may have no previous knowledge of their idioms. It does not matter, after some time you will learn some words and may even *constitute incrementally a dictionary* including not only new words but also new concepts. Trying to get help from locals is the key to successful exploration. Locals can give you plenty of useful hints: how to use local tools, how to move around, how to get more information, etc. *Using local transportation systems* is fundamental to back-packing. Maybe you can walk, ride horses, and/or swim, but during your trip you may also need to learn camel-riding, skiing, etc. Though it may take you some time to learn, *your ability to adapt to the local environment is fundamental.*

Finally note, that if you walk in an active country; *you may interfere with your environment*, so you have to expect some strange behavior. Exploration may represent even some danger. Back-packing is not always fun, but if you take time and have a great level of adaptation, *you will clearly discover new things, new places, new civilizations.*

3.2. The auto tour

Travellers who want to discover a new country can alternatively contact a tour operator. Some standard packages include tourist routes in tourist countries. If you want something more specific, for instance to learn more about a specific theme, you can try to make some specific arrangements with the tour operator. They could prepare the tour for you (maybe by sending back-packers to explore the country). However, this *preparation could take*

considerable time and effort. You will not find any travel agencies, if you are the only traveler who want to go there.

Let's assume however that after a few negotiations, you purchase an auto-tour from a travel agency. You will receive a car-rental voucher with travel information: a brochure and a set of documents corresponding to the theme you've selected. All these documents have been translated for you by someone else and are supposed to contain everything you need. During the trip, you will never communicate with locals; instead you read the documents, which are a partial representation of the country. *If for some reason you need some additional information, it is complicated:* you have to contact the travel agency and wait for additional documents to be sent (if they already exist). This contrasts with the back-packing approach, where information is obtained on demand, only when required.

The auto-tour is also characterized by the use of a *single mode of transportation*: the car you rent. You can go where you want and when you want, but only if there is a good road. If you want to learn something about a civilization living in the far country-side, you can go to the museum, but once again, *this is not the world, but just a representation of the world previously elaborated with a given set of themes in mind.*

With the auto-tour approach, you do not interfere with your environment and you travel safely. You can see new landscapes, but *you cannot expect to discover new kinds of things and new concepts* that people like you have never seen before.

3.3. The limousine

If you travel for business and have limited time, you need some efficient way to travel. You don't want to worry about anything else but your job. Take a plane. At the airport, get a luxury limousine. Even if you've never been to this city, don't worry: *everything has been arranged to provide the greatest level of comfort* to businessmen like you. *You don't need a dictionary* since only a few concepts are involved: hotel, convention center, airport, etc. If you don't really know where to go, don't worry: the chauffeur will bring you automatically to one of the best hotels in town. Behind the window, you can still discover some new things during the trip, but this way to travel soon becomes a nice and comfortable routine that allows you to do your job. *Unfortunately less than one percent of the world can be "explored" like that.* Moreover, if you are not able to prove direct benefits from you travel, your company will not cover the cost of this luxury service.

4. Three approaches to software exploration

The (software) world is full of software artifacts sold by large software companies or available as open-source and freeware. The interest of "software exploration" has been claimed for a long time, but as shown below, this term can encompass a wide range of rather different goals and activities.

4.1. Software land

To provide a useful discussion, it is necessary to distinguish first different levels of description. The number of levels and the name of each level varies from field to field. In the context of this paper we will use 4 levels, just like in the UML standard [36]: the instance level, the model level, the meta-model level and the meta-meta model level. Simply put, each level is a language that describes the level immediately under it. Conversely each element of a level is an instance of an element immediately superior to it.

To illustrate these abstract concepts, let us assume for instance that the goal is to explore a banking application software. Table 1 represents the different levels and gives alternative vocabulary.

Table 1. 4 levels of information

Levels	Examples
Meta-meta model (data model)	<i>Relational data model:</i> relation, tuple, attribute <i>ER data model:</i> entity, attribute, relation <i>Object-oriented model:</i> class, association, inheritance, composition, aggregation, ...
Meta-model (domain [13], schema [48], model [37], conceptual model [6], meta information)	<i>C language:</i> File, Function, Variable, Type, Macro, Expression <i>Java language:</i> Package, Class, Field, Method, Inheritance
Model (data)	Bank, Client, Account, withdraw, ...
Instance	joe, acc#23021, withdraw execution #12034

The **instance level** describes possible states and executions of the software. At this level we may find for instance the entity representing the *client* "joe", its *account* "ac#20132". This level can be safely ignored if the execution of the software is never considered.

The **model level** describes the software itself. If the program is written in C we might find the "withdraw" *function* and the "account" *type*. In the case of Java, we might find the "withdraw" *method*, the "Account" *class*. Software exploration tools typically deal with this level and represent static information as "software facts".

The **meta-model level** describes the programming language used to represent the software. For instance in the

case of the C language, we will find the following *types of entities*: “File”, “Type”, “Variable”, “Macro” and “Function” [15]; and the following types of relation: “Include”, “Call”, etc. In the case of the Java language it will be “Package”, “Class”, “Method”, “Field”, etc. and “Inherits”, “Call”, etc.

At **meta-meta model level** we find the necessary concepts to describe the meta-model. In the paragraph above we have implicitly used the entity-relationship (meta-meta) model, since we define the concepts in terms of entity types and relation types. Other potential candidates include the relational data model (“relation” and “tuples”), object models (“class”, “inheritance”, etc.), and so on.

Most practitioners might consider these concepts too abstract to be useful in an industrial context. Describing the complexity of real software countries requires these levels to be distinguished. A similar approach is described in [49] in the context of a reverse engineering project in a large software company.

4.2. Software limousines

Today’s integrated development environment includes many specific tools that could be assimilated as “software limousines”. They enable software engineers to travel efficiently within software structures and concentrate on their job with a maximum level of comfort. This includes for instance class browsers, source code browsers, etc. Facts about software are extracted transparently. Visualization is of great quality thanks to specific layout algorithms, highly optimized rendering, etc.

Unfortunately this approach suffers from the limitations of the limousine approach. You can go only to a limited number of places. Typically a limited number of views is offered, only for software products written in a given set of programming languages. For instance, some tools are restricted to the exploration of the inheritance relationship between Java classes, others to the control-flow structure of C programs, etc. *The meta model is fixed and hard-wired.*

Maybe less than one percent of the software world can be explored with standard software limousines. As pointed out in [19], almost only traditional programming-in-the-small artifacts can be explored in this way. If you are brave enough, you can still develop your own limousine to travel in your own domain. For instance, in the context of Dassault Systèmes, we developed the OMVT tool, a software limousine to explore DS components [10][23]. However, *the cost of development of software limousines is high.* Moreover, without extensive traveling experience in a software country *it is very difficult to know in advance which views will be useful to software engineers.* So if you want your limousine service to be used daily, you have to be prepared for strong evolution in order to fulfill the

requirements of software engineers [23].

In any case, currently *in software industry the only accepted way to travel is using software limousines.* This is not surprising since most software engineers have to concentrate on their jobs. They will not use such exploration tools unless they can give rapid answer to specific questions.

4.3. Software auto tours

The limitations described above have given rise to the appearance of “software travel agencies” providing you “software auto-tours”. Over the last decades the Rigi system [13] developed at the University of Victoria is one of the best representative of this approach. Its flexibility and availability have been particularly appreciated in the research community; many software travels have been organized thanks to it. This environment is “domain-retargetable”: the meta-model (referred as the domain) is not fixed, you can define a new one¹.

As a software explorer, you may want to travel in a standard country using a standard auto-tour provided with Rigi. For instance, if you want to explore the architecture of the C software, the trip can start immediately; Rigi will offer you a pleasant journey. Let’s assume however, that you want to explore a non-traditional country and let’s see what preliminary preparations are required before the exploration can start. As shown below this is not a simple process.

4.3.1. Elaborating a Meta model. First you have to define the theme of your auto-tour, that is the meta model (the domain). Roughly speaking, that means you have to think about which types of entity and relation you want to see during your trip. This step could be quite easy if you have a clear idea of what kind of information you want to see. Simple meta-models typically contain a dozen entity types and relations (e.g. [15][17]). Things could get significantly more complicated, if you plan an in depth exploration of a complex yet standard country such as C++ land [38]. Things will get even worse if you’ve never been to the software country. In this case, you do not really know in advance what is worth seeing since you don’t know what exists. In this case meta-model elaboration is only possible through

1. Other environments such as PBS [11], GUPRO [12], Shrimp [14] or Moose [1] provide similar features and many others. While PBS, Rigi and GUPRO are *domain-retargetable*, Moose is *domain-extensible*. This environment is dedicated to the exploration of object-oriented software and is based on a core meta-model describing common features of object-oriented languages. This core meta model can be *extended* to add additional information specific to a language for instance. However, to simplify the discussion, in the context of this paper, we will focus on Rigi to illustrate the auto-tour approach; especially since it is the best-known tool and is freely available.

extensive discussions with local authorities, a careful study of the documentation they may provide, and maybe a close examination of existing software. In the context of DS, it took us a few months to define a suitable meta-model [10]. Environments like Rigi, do not offer any help in the meta-model elaboration phase.

4.3.2. *Selecting an Exchange Format.* In the auto-tour approach all information about software must be written in an exchange format. For instance, Rigi and Shrimp read facts expressed in Rigi Standard Format (RSF) files, PBS reads TA files, Moose is based on Famix, GUPRO on GXL, etc. In the last years significant effort has been devoted to defining standard exchange formats between software engineering tools. This includes for instance, XMI for the exchange of UML models [36], and GXL in the reverse engineering community [39]. While these approaches are very promising and must be strongly encouraged and supported, they may not solve all problems. Some formats will become the equivalent of Esperanto; only few ones, if any, will become in the long term an international language such as English. What is more, while many English-speaking people implicitly assume that the world can be explored using their own language, they are often disappointed when travelling in foreign countries... Beside the problem of choosing a format, the issue is how to get information in that format?

4.3.3. *Using, building or wrapping an extractor.* Once the meta-model has been elaborated and a suitable format chosen, you still have to find a way to extract information from software artifacts. In the context of reverse engineering, such a tool is commonly called a (software fact) extractor.

If you plan to visit a reasonably touristy software country, you may find the extractor you need. For instance, if you are interested in visiting C++ land, you will find CPPX useful [41]. In some very popular countries, you will have to choose from various extractors available. In practice this is not an easy task. You will discover that different extractors often give different results for the same piece of software and that nothing indicates which extractor to trust. Often various extractors are complementary, and integrating extractors is an open issue in practice [40][48].

If such an extractor does not exist in the country you want to visit, you can try to convince local authorities to include some piece of code in one of their working tools in order to gather the information you need. But this could be far from easy in an industrial context. How will you convince them to do this if you have nothing to show in return?

At the end, as a brave software explorer, you will probably have to build your own extractors to demonstrate

the usefulness of software exploration. This could be quite difficult. You may either develop a new parser, but this is usually not reasonable, or try to modify the source code of an existing tool to include code generating the information you need. This approach raises several issues. You first have to understand the code of the extraction tool¹. Then you have to extend this code to generate the structure you need. This can be quite difficult [41], since getting the necessary information may require complex recursive traversals of unfamiliar software structures. In some cases, it could be very difficult to ensure that all required entities have been traversed. To further complicate your task, some formats are quite complex and it is sometimes impossible to generate the required information in a single pass, thus requiring multiple structure traversal. Finally, with this approach you will introduce instability in the extractor and may have to synchronize your modifications with the evolution of the extractor...

4.3.4. *Extracting actual information.* Finding or building the extractor is probably the most complicated part in the preparation process. However, you cannot really evaluate the quality of an extractor without trying to extract actual information from a piece of software. With the auto-tour approach, all information that the explorer could need during his travels must be extracted and represented explicitly before the beginning of the exploration. Extracting all this information could require a couple of hours for a large software. Since it is not possible to make any assumption about explorer moves, a large amount of information is extracted. Files soon became huge (several mega bytes!) especially when using formats like XML. In a system like Rigi, these files are read at the beginning of each exploration session and long loading delays prevent opportunistic use of this tool.

4.3.5. *Enjoying the auto-tour and getting frustrated...* Fortunately, the long preparation process described above must be applied only once per meta-model, and you might find somebody else to organize all this for you. Once an auto-tour has been defined it can be reused by many different software explorers. Thanks to a tool like Rigi, explorers are able to travel along any relations that have been defined in the meta model, in the order they want. They can discover new software landscapes. However, sooner or later, the explorer will suffer from a lack of information during the auto tour. Remember that this approach is based on the definition *a priori* of the meta model. During a session an explorer often changes his mind

1. Interestingly, this is a software exploration task, since you have to explore the extractor [27]. So you have the opportunity to try the effectiveness of your research in a very concrete context...

and may want to learn more about a specific portion of the code requiring to explore a lower level of granularity. For instance an explorer might want to explore the detail of a Java method, and may be its bytecode, after having identified a faulty method in a class hierarchy. Unfortunately this kind of exploration will not be possible if the type of entity has not been defined in the current meta-model. The problem is not only adding necessary types to the meta-model, but also modifying the code of the current extractor again, or worse finding a new extractor.

Actually, in the context of DS, we tried the auto-tour approach and we evaluated the Rigi environment [24]. *While the auto-tour approach is well suited to rather stable and touristy software country (e.g. C/C++ programming), it shows its limits when exploring unexplored and evolving software country.* This led us to the back-packing approach.

4.4. Software back-packing

Software back-packing is quite a novel approach to software exploration. It contrasts with software limousines in that it is not restricted to a very specific part of the world. It contrasts with auto-tours since it does not require long preliminary preparation. It is suited to *extreme exploration* in the furthest bounds of the software world (actually as pointed out before, only small parts of the software planet have been explored). *The meta-model is elaborated along the way, new source components (extractors) are integrated interactively and new concepts are discovered on the fly. The philosophy of the software back-packer is to adapt to the world, while the philosophy behind the auto-tour is to adapt the world to the traveller.*

The attentive software back-packer will notice that software countries are not only full of software artifacts, but also full of local tools used to manage these artifacts. Actually, there are plenty of existing sources of information in a large software company: a configuration management system, a database of bug reports, etc. *Why not try to communicate directly with local sources of information?* Our belief is that the key is to be able to adapt to different meta-models *and* meta-meta models. Obviously, you can hardly expect local workers to speak in terms of such abstract concepts. But their work must not be underestimated: they use many ways to structure the software they produce, at different levels of abstraction. You just have to find a way to recover this information. In particular, during exploration, *the explorer should be able to discover new types of things, but also define new types or refine existing types. This is meta-model (re)discovery.*

Here an analogy is required to show that this concept is practiced every day by many people, though not to explore software.

Let's consider for instance, that you are a novice user and that you are using a file browser for the first time. By just clicking on filenames and observing the result, you will soon discover that some entities are suitable for navigation (directories) while others display their contents (plain files). Eventually, you will discover through careful examination of filenames, that they convey type information: the characters after the dot seem to reflect the type of the file! After this great discovery, you might decide to record each type of file you discover. In this way you constitute useful meta information. Some day, you may discover a new type of file that your favorite browser cannot open (for instance a zip file). With the clear intuition that there is something either inside or behind this file, you will probably ask for external help. If nobody is there, you may look for something in the browser. You will eventually learn how to change the mapping between file types and the application that opens it. Binding the zip type to an application like Winzip will allow you to continue the exploration of this fascinating world. After a few years' experience, you will know how to download new applications on internet to open new types of files. In this exploration scenario, the user not only discovers new information, but also constitutes and refines substantial knowledge about meta-information and even meta-meta information. This is the kind of exploration we think is required to explore unexplored software country.

5. Software back-packing with G^{SEE}

In the previous section we have shown the limits of existing approaches to software exploration and described the philosophy behind software back-packing. While this approach is quite general and obviously requires further investigation, we already have made some important steps in that direction thanks to G^{SEE} [20].

5.1. The G^{SEE} back-pack

Before taking the plane, let's check what can be found in the G^{SEE} back-pack. Its content has been defined and refined during our previous software travels. We often add old ones and improve existing ones. This is really the nature of a back-pack. Actually the content of the back-pack is organized with different levels; tools of common use being on the top to be at hand.

At the bottom of the back-pack, there is an *object-oriented framework*, that is a well organized set of small classes. Describing this OO framework is outside the scope of this paper. In [20] the *Successor* interface is described. This interface is used to explore software structures including databases, data structures, APIs, batch systems, in a transparent way. Actually, this interface is the equivalent

of the Swiss-knife in the context of software exploration.

The medium level in the G^{SEE} back-pack corresponds to a *component framework* (based itself on the OO framework). Thanks to this layer new applications can be assembled interactively from existing components. This level is based on our work in component-based software engineering [46] and in particular on the Beanome language [45], an extension of the JavaBean component model [29]. The back-pack contains a large set of visualization components making it possible to produce powerful visualization very easily [31]. One important benefit of components is that they can be connected interactively to form new applications. We are currently working on the elaboration of the G^{SEE} builder similar to the BeanBox [29] but providing extended capability. Actually G^{SEE} shares a lot of features with meta-case environments and in particular we follow an approach similar to JKogge [47].

On the top level of the back-pack, we keep some small directly usable tools. These tools were developed during previous exploration campaigns. This includes in particular the G^{SEE} *Viewer*, which provide a rudimentary yet powerful way to define new views on arbitrary software structures, the G^{SEE} *Interpreter*, which allows the explorer to evaluate arbitrary expressions of the G^{SEE} *Query Language*. Recently, we have also included the G^{SEE} *Walker* to allow simultaneous exploration of the model and the meta-model. All these tools are built on top of the framework; in the begining there were built only to demonstrate the power of the approach. These tools are rudimentary (some of them are less than 100 lines of code), but they have proved very useful in practice. Thanks to them it is possible to get results very rapidly. To illustrate software back-packing, we describe below a typical exploration with the G^{SEE} *Viewer*.

5.2. Landing and getting authorizations to travel...

Everything starts with the will to visit a new software country¹. Get the back-pack. Install it (on the computer, not on your shoulder) and start for instance the G^{SEE} *Viewer* (Figure 1). At the beginning you can explore nothing. The first issue in software back-packing is to find some sources of information. Fortunately G^{SEE} is based the *Repository* concept. This abstraction makes it possible to connect to most common sources in a transparent way.

For instance to explore DS' software as in [10][23], then it is enough to open a repository with a name looking like "*os:C:\DSDATA\IDS12031.odb*" (figure 2). The prefix *os* indicates to load the wrapper corresponding to *Object-Store* OO DBMS [32], and the rest of the string will be interpreted by the wrapper². In this case, this is the localization of the

1. You may need to realize an assessment in a software company; or you may have just received a new software or extractor.

database itself. It is very important to stress at this point that G^{SEE} does not contain a single line of code specific to Object Store DBMS, nor to the Dassault System database³.

Similarly, an arbitrary Java program can play the role of repository and can be loaded dynamically. In this case the name of the repository may look like "*java:http://www.foo.edu/myextractor/x.jar*". For instance, in [20] a scenario is given where the JAssistant tool [35] is downloaded from internet and used immediately to extract facts on Java programs. In a similar way, you can also load RSF files for instance. We plan to add support for GXL. Note finally that various repositories can be opened at the same time and that data stored or extracted on the fly by different repositories can be accessed transparently.

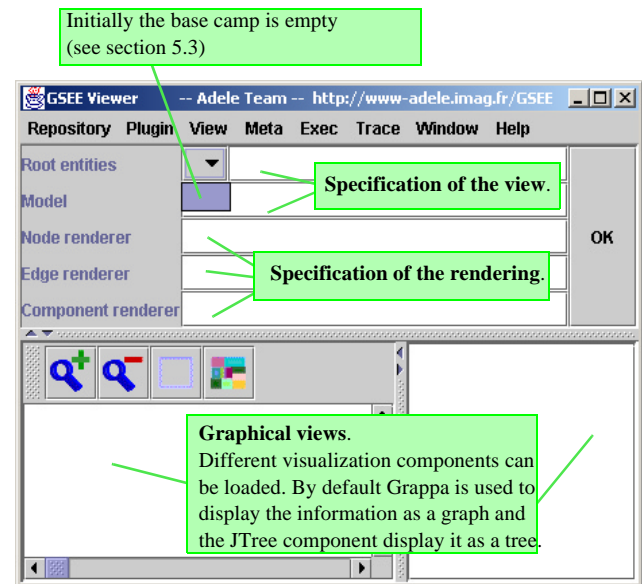


Figure 1. Initial interface of the GSEE Viewer

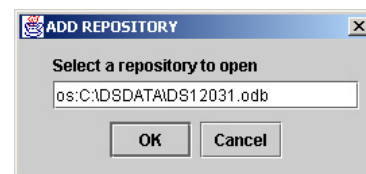


Figure 2. Opening an object store database

2. We use an architecture similar to JDBC [44], the standard API to access any relational databases in a totally transparent way, but with a data model (a meta meta model) more suited to software exploration.

3. Object Store is a commercial DBMS that has been used to implement the software limousine OMVT [10][23]. To explore this source of data with G^{SEE} , a wrapper to Object Store has been written in Java. It is only 61 lines long and can be used for any Object Store database, not just DS software.

5.3. Setting up a base camp...

Once the connection is established, the first step is establishing a base camp. The problem is to identify a set of points from where each excursion can start. Some repositories will automatically provide a lot of such points. For instance, repositories based on RSF files give the opportunity to start from any entities since they are all enumerated in these files. Repositories based on object-oriented databases will allow you to start only from “root objects” [32][42]. For instance, opening the DS’ repository (Figure 2 on the previous page) automatically gives access to the root entities stored in the database. When the repository is based instead on some piece of code (e.g. a Java repository or a batch system repository), establishing the base camp may require more work. Due to space limitations this step is not described in this paper but finding entry points can be automated or at least assisted. Our experience suggests that establishing a base camp in a foreign country is usually a simple task [27]. And you only have to do it once.

Let’s assume in our case that, in addition to explore DS software structures, we want to build a file browser. The problem then is how to get a file. It is enough to add a *Finder*¹ to the base camp (Figure 3).

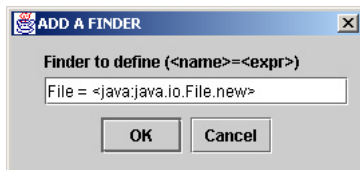


Figure 3. Adding a finder for files

If we also want to build a Java class browser, then add a *Finder* to retrieve Java classes (Figure 4). All this at the same time, during the same trip.

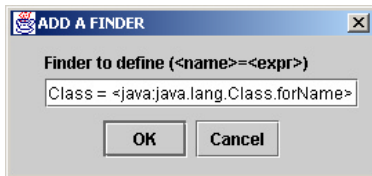


Figure 4. Adding a finder for Java classes

Actually, to establish a base camp we found the concept of *pair exploration* useful, that is the exploration of a country by a team of two persons: a specialist of the G^{SEE} back-pack and a local who knows the country well. The local will introduce you to other locals and to the tools they use. In particular the local will give you advice about available extractors and help you in establishing the base

camp. Once established you usually become autonomous and able to start exploration.

5.4. Getting to the first place...

Every exploration starts from the base camp. Depending on the capability offered by the repository, the explorer will either have to supply a string to retrieve a given entity or pick from a list of existing entities if the repository provides this facility. The G^{SEE} Viewer tool automatically adapts its user interface to the facilities offered by the repository. For instance, in the figure 5 the pull down menu lists the types of entities available.

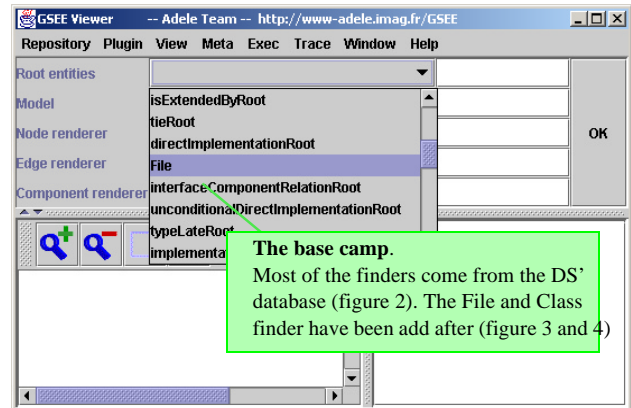


Figure 5. The base camp

To explore the file system, just enough to select “File” in the menu and then supply a filename in the field. G^{SEE} will retrieve the resulting object. This can be an Java object, a data structure, a portion of an XML file, or anything else. It does not really matter. G^{SEE} will display the resulting object in the bottom of the window.

5.5. Looking around and reading the signs...

Before making the first step, you have to look around to choose a direction. The entity you’ve got may be accompanied by some kind of meta information. For instance it may have some type and/or may have some relations (*Successors*) attached to it. G^{SEE} tools are able to recognize the presence of this kind of meta information depending on the meta-meta model associated with the repository. Typically, if the object is a Java object G^{SEE} will naturally use Java introspection. If the entity comes from an RSF file, it will use the Rigi type system, etc. In some situations the explorer might also want to define or refine the notion of type as described in section 4.4. Anyway, the important thing is that wherever you are, you can still look around you to see if there are some signs indicating the directions that can be taken from there.

1. This concept is used in the Corba Component Model (CCM) [30]

Figure 6 shows a popup menu displayed over the file that has been just retrieved. It indicates on the first line the type of the entity (here “File”) and the list of directions that can be followed (here the methods attached to this Java object).

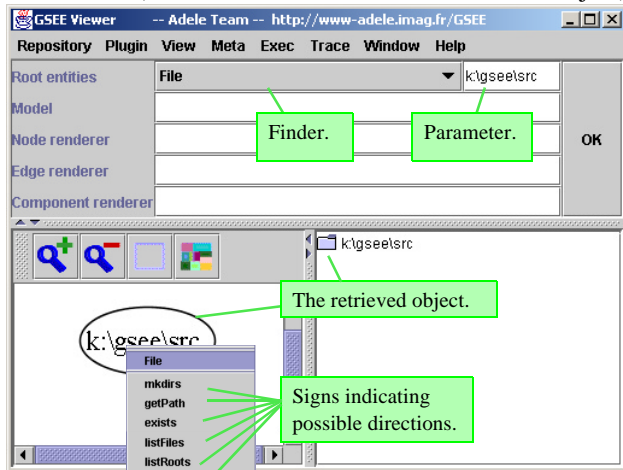


Figure 6. Looking around and reading the signs...

5.6. Making the first step...

Note, that just like in a foreign country you may not really understand the signification of the signs, especially when those signs were placed by locals who never imagined that a software tourist would come here and look at them: if the repository is a RSF file, you may not understand the meaning of the name of relationships ; if the repository is actually a program, the name of the methods may not be very explicit since most programmers do not expect other people to introspect their code.

Anyway, even if you do not really understand the sign you can still follow the direction indicated by the sign... After all, pushing the limit of the unknown is a characteristic of exploration. That is the essence of software back-packing. Following the sign, that is executing the *successor*, will typically invoke a method, traverse a data structure, or execute a script. In fact, you probably don't have to know the details. Be brave, just make a step ahead to see what happen...

In general you will get a collection of entities: the successors of the entity from where you started. For example if you choose to follow the sign “listFiles” you will get a collection of files¹. Interestingly these new entities may be of various types, or of unknown type. Maybe information about type was clearly provided on the sign you've just followed, but this depends on the meta-meta model being used. Even if it was the case and if a given type

1. Actually listFiles is a method of the class java.io.File. It returns an array of file, not a collection. But this does not matter, G^{SEE} automatically makes the necessary conversions behind the scenes.

was clearly indicated, you can still discover entities of subtypes if the meta-meta model allows polymorphism. For instance, the entity accessible through “listFiles” can be of any File subtype. What it is true, is that in general you will discover a bunch of new types along your way. But don't worry, G^{SEE} is collecting meta-information for you silently.

5.7. Walking...

As usual the first step is the most impressive. But, step by step, you will soon be walking in this totally new software country. To ease this process you can open the G^{SEE} Walker as shown in the figure 7 (the initial point here is the java class named *javax.swing.JButton*).

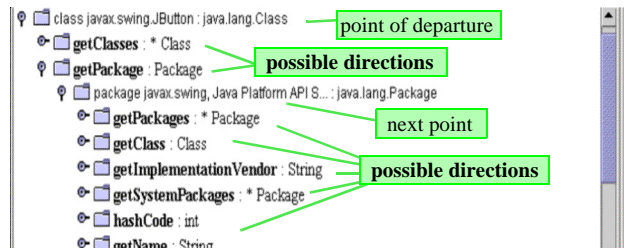


Figure 7. Walking ...

This very simple tool will allow you to follow, in a tree structure, different directions at will. A fundamental point here is that the information is computed only when required: if you don't open the door (click on the Successor node), then you will never enter there (the Successor will never be executed).

On the one hand, this property makes it possible to walk around in extremely wide software structures without the requirement that all doors have been previously opened for you before (this is one limitation of auto-tour as pointed before). For instance, you can browse a file system with millions of files, or a software with a thousands of classes, with no problem at all: only the files or classes you visit will be analyzed on demand in a transparent way.

On the other hand, leaving a door closed may also keep you away from danger. It is probably better to not open a door with a "destroy" sign on it... As pointed out before, the back-packer is sometimes walking directly in an active software country, and his presence may have an impact on the environment. This contrasts with walking in a museum (see Section 3.2) where all objects are dead. Here you are walking in an live country. Care should thus be taken, but our experience shows that in practice solutions exists to ensure a good level of security [27]. In particular, a dangerous *Successor* could be marked and eliminated from the meta-model. This is another example of meta-model elaboration and refinement.

5.8. Running and getting to the top of a hill...

Walking and looking at all signs before each step rapidly becomes very boring. You may want to run in a particular direction without stopping at each intersection. The G^{SEE} Viewer tool will help you. Just specify which paths you want to follow, and it will automatically traverse the country for you. G^{SEE} provides a large set of operators to compose *Successors*. Running around the country enables you to get to the top of a hill and to get an overall view. In the G^{SEE} backpack you will find a large set of sun glasses (the visualization and rendering components). After some adjustments, you can really enjoy a pleasant view. For instance, in [20] a post card from Java land is provided. It shows more than 600 classes from the swing and awt packages. Figure 8 below displays a postcard from Dassault Systèmes software country with more 796 component interfaces.

What is the difference? Well, they come from two totally different software countries, with totally different meta-models (software written in Java in the first case, and software written in C++, with a component technology in the second case), totally different representations (information extracted on the fly vs. facts stored in a database). What are the common points? Well, thanks to G^{SEE} , it took only a very limited effort to get there. Both views result from an exploratory approach. In particular both views rely on the use of metrics that have been defined on-the-fly during exploration.

5.9. Taking a rest and summing up...

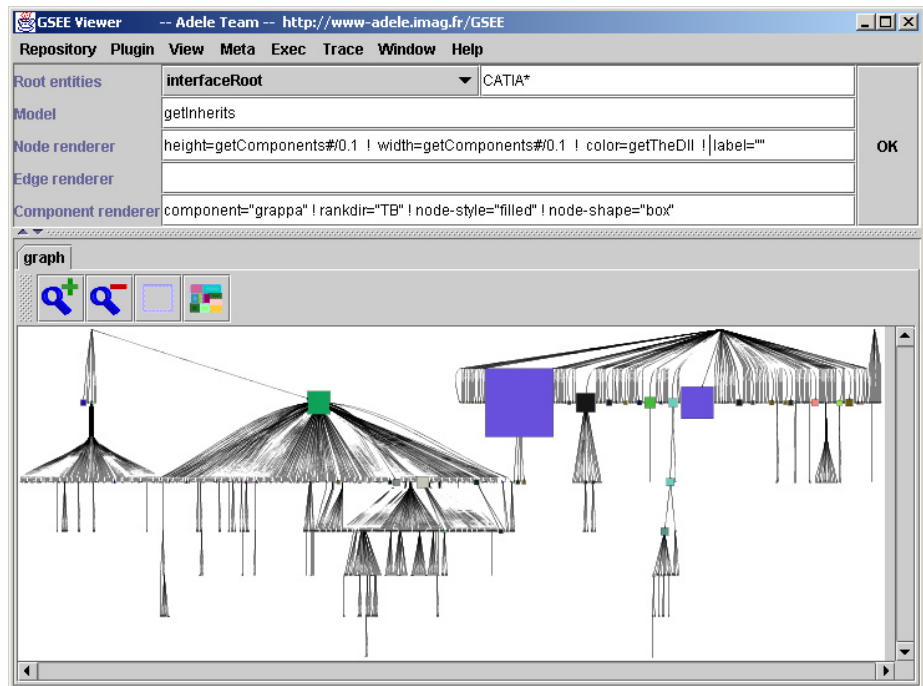
At the end of the day, you've probably seen too many things. You may have discovered many places and many software entities, but also many new types of things. You have to take a rest and remember what really happened in that exciting journey. Fortunately, during your travel much information was gathered for you, thanks to different metrics and recording tools. In the G^{SEE} back-pack you will find tools that summarize the activity of the day. In fact, at any time you can leave the exploration of software entities and swap to the exploration of the current meta-model.

Figure 9 on next page shows a view on a small subset of the DS' meta model described in [10]. This view has been automatically generated. The elements in this view are types, not entities. As the reader can see, the same tools can be used (here the G^{SEE} Viewer) to explore the meta-model¹. Simply put, the meta-model is just like a sophisticated kind of dictionary or a key for a map. By looking at the key you discover the essential concepts in a software world as well as the relationship between these concepts. Actually, we found during our travels that meta-model elaboration, exploration and refinement are a very interesting research issues. We already have gained experience in this domain, but this topic is too large to be discussed in the context of this paper.

1. To be honest this could be quite disturbing for the novice explorer, but at this moment we do this to limit development effort. Note that we can also explore, in the same way, the meta-meta model... Actually it is so cheap to build new views with G^{SEE} that we are constantly discovering new ways to explore software and new research issues.

Figure 8.

A post card from DS' software country ...



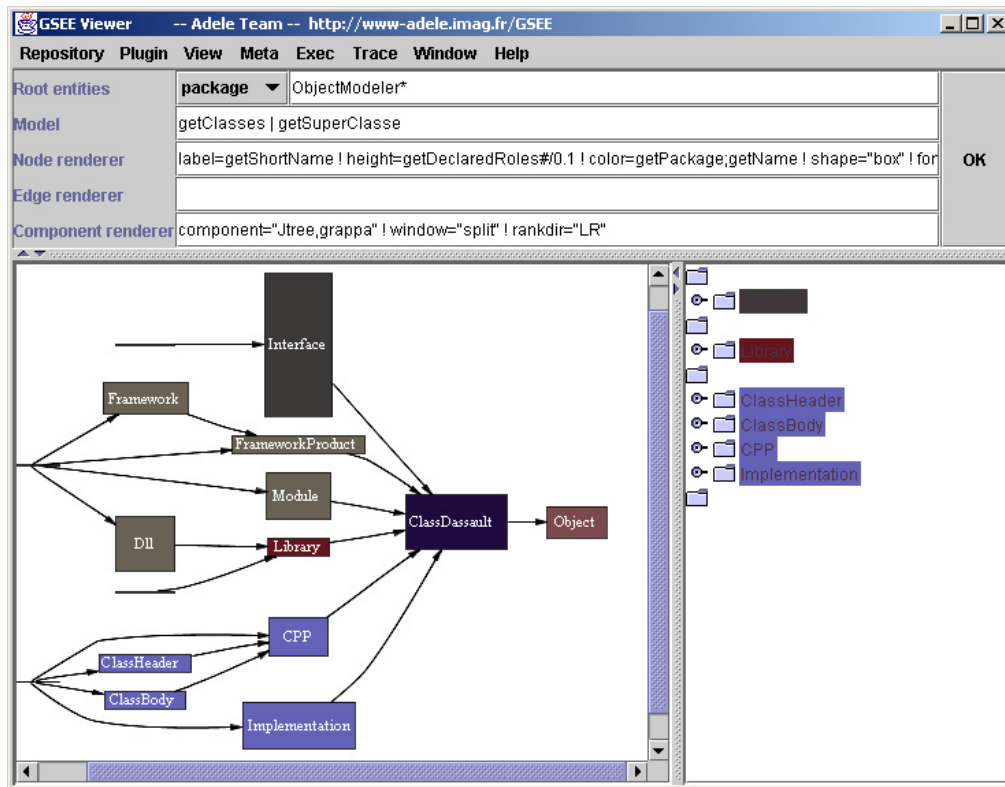


Figure 9. Summing up and thinking a little bit...

6. Conclusion

There are many ways to travel around the world, but they are not necessarily equivalent since they do not correspond to the same needs. In this paper, we have claimed that current approaches to software exploration are limited in scope since they require a significant amount of knowledge about the software country to explore.

Does software back-packing differ from other software exploration techniques? We believe so. It cannot be compared with software limousines since back-packing is by no means restricted to very specific areas of the software world. Software backpacking also significantly differs from auto-tours. Auto-tours are based on the availability of an explicit representation of the world in a given format, but do not provide solutions to construct these representations if they do not already exist. Moreover, auto-tours are based on the previous elaboration of a suitable meta-model, but do not offer help in this respect. This contrasts with software back-packing, which requires almost no preliminary preparation. New sources of information can be added interactively and meta-models can be discovered and recovered on-the-fly. It should be noted however, that all techniques of exploration are complementary since they correspond to different needs.

Is software exploration useful in the software industry? Just like many other researchers in this field we believe so, but it still remains an open question [9]. We must admit that our experience suggests that in industry, software exploration is often assimilated to tourism. You know, in any country local workers who are very busy just see explorers as tourists: they do not really understand what they do and why they take pictures of things that seem to be ordinary-life. The explorer should *never* underestimate the knowledge that an old local can have about his country. However, the local should never assume that he is *always* right. To really convince software engineers and management of the benefits of software exploration, you first have to show them either that their beliefs do not correspond to the reality, or things that they have never thought before. If they know well their country, then you must be very agile; you must be able to get useful information very quickly and in very a cheap way. G^{SEE} could provide you with such flexibility. Copies of the back-pack will be available for free. So enjoy it to discover new software horizons...

See <http://www-adele.imag.fr/GSEE>
and get involved in the G^{SEE} odyssey!

7. References

- [1] S. Ducasse, S. Demeyer, editors; "*The FAMOOS Object-Oriented Reengineering Handbook*", Univ. of Bern, Oct. 1999.
- [2] P. Eades, K. Zhang, "*Software Visualization*", World Scientific Pub., 1996.
- [3] S.R. Tilley. "A Reverse-Engineering Environment Framework.", Technical report CMU/SEI-98-TR-005, 1998
- [4] S.R. Tilley. "Domain-retargetable reverse engineering". Ph.D. Dissertation, University of Victoria, 1995
- [5] R. Kazman, S.J. Carrière; "*Playing Detective: Reconstructing Software Architecture From Available Evidence*", Tech. Rep. CMU-SEI-TR-010, Software Engineering Institute, 1997.
- [6] J. Ebert, M. Kamp, A. Winter, "GUPRO: A Generic System to Support Multi-Level Understanding of Heterogeneous Software", Fachbericht Informatik 6/97, Universität Koblenz-Landau, Institut für Informatik, Koblenz, 1997
- [7] A.O. Mendelzon, J. Sametinger. "Reverse Engineering by Visualizing and Querying", Software : Concepts and Tools" Vol. 16/4, 1995
- [8] 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.
- [9] S. Bassil, R.K. Keller, "*Software Visualization Tools: Survey and Analysis*", International Workshop on Program Comprehension, (IWPC'01), 2001
- [10] J.M. Favre, F. Duclos, J. Estublier, R. Sanlaville, J.J. Auffret, "*Reverse Engineering a Large Component-based Software Product*", European Conference on Software Maintenance and Reengineering (CSMR'01), March 2001
- [11] R. Holt et al, PBS, <http://www.turing.toronto.edu>
- [12] J. Ebert et al, GUPRO, <http://www.gupro.de>
- [13] H.A. Muller et al, RIGI, <http://www.rigi.csc.uvic.ca/>
- [14] M.A. Storey et al., <http://www.csr.uvic.ca/shrimpviews/>
- [15] Y.F. Chen et al. CIA/++, <http://www.research.att.com/~ciao/>
- [16] Spool Project, <http://www.iro.umontreal.ca/labs/gelo/spool/>
- [17] Imagix Software, <http://www.imagix.com>
- [18] Software exploration, <http://www.software-exploration.com>
- [19] J.M. Favre, "*Understanding-In-The-Large*", 5th International Workshop on Program Comprehension (IWPC'97), 1997.
- [20] J.M. Favre, "*G^{SEE}: a Generic Software Exploration Environment*", International Workshop on Program Comprehension (IWPC'2001), May 2001.
- [21] J.M. Favre; "*Preprocessors From an Abstract Point of View*", Int. Conference on Software Maintenance (ICSM'96), 1996
- [22] J.M. Favre; "*A rigorous approach to the maintenance of large portable software*", European Conference on Software Maintenance and Reengineering (CSMR'97), March 1997
- [23] R. Sanlaville, J.M. Favre, Y. Ledru, "*Helping Various Stakeholders to Understand a Very Large Software Product*" European Conference on Component-based Software Engineering (CBSE'01), 2001.
- [24] S.T. Nguyen, J.M. Favre, Y. Ledru, J. Estublier; "*Exploring Large Software Products*", (in french), 13th International Conference on Software and Systems Engineering and their Applications (ICSSEA'2000), Dec. 2000.
- [25] J. Estublier, J.M. Favre, R. Sanlaville, "*An Industrial Experience with Dassault Systèmes Component Model*", Chapter in [26].
- [26] I. Crnkovic, M. Larsson, "*Builiding Reliable Component-Based Systems*", Archtech House publishers, to appear, 2002
- [27] J.M. Favre, "*Exploring java land through the exploration of java extractors*", in preparation.
- [28] J. Estublier, R. Casallas, "*The Adele Configuration Manager*", Chapter in Trends in Software, J. Wiley and Sons, 1994
- [29] JavaBeans Specification. <http://java.sun.com/products/javabeans/docs/spec.html>
- [30] "*CCM: Corba Component Model*", OMG, August 1999
- [31] J.M. Favre, "*A Flexible Approach to Visualize Large Software Products*", ICSE Workshop on Software Visualization, 2001
- [32] <http://www.objectdesign.com/products/objectstore.html>
- [33] 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>
- [34] M. Dahm, "*Byte Code Engineering with the JavaClass API*", Tech. Report, TR-B-17-98, Univ. Berlin, Dec. 1998.
- [35] A. David, "*JavaAssistant 1.6, On-the-fly Class Browser*", <http://www.docs.uu.se/~adavid>
- [36] UML Standard, V1.4, September 2001
- [37] D.E. Perry, "*Software Interconnection Models*", Proc. of the 9th Int. Conf. On Software Engineering, IEEE, March 1987.
- [38] R. Ferenc, S.E. Sin, R.C. Holt, R. Koshke, T. Gyimothy, "*Towards a standard schema for C/C++*", Working Conference on Reverse Engineering (WCRE'01), 2001
- [39] GXL, <http://www.gupro.de/GXL>
- [40] J. Michaud, M.A. Storey, H. Muller, "*Integrating Information Sources for Visualizing Java Programs*", International Conference on Software Maintenance (ICSM'01), 2001
- [41] CPPX, "*Open Source C++ Fact Extractor*", <http://swag.uwaterloo.ca/~cppx/>
- [42] R.G.G. Cattell et al, "*The Object Database Standard: ODMG 2.0*", Morgan Kaufmann Publishers, ISBN 1-55860-463-4, 1997
- [43] "*JavaTM APIs for XML Processing (JAXP)*" <http://java.sun.com/xml/jaxp.html>
- [44] JDBC, <http://java.sun.com>
- [45] H. Cervantes, J.M. Favre, F. Duclos, "*Hierarchical Composition of Java Beans with Beanome*", submitted to the Workshop on Software Composition, ETAPS, Grenoble, France 2002.
- [46] J. Estublier, J.M. Favre, "*Component Models and Component Technology*", Chapter in [26]
- [47] J. Ebert, R. Sutenbach, I. Uhe, "*JKogge: a Component-Based Approach for Tools in the Internet*", Proceedings of STJA '99, Erfurt, 1999
- [48] I.T. Bowman, M.W. Godfrey, R.C. Holt, "*Connecting Architecture Reconstruction Frameworks*", Proc. of the First International Symposium on Constructing Software Engineering Tools (CoSET'99), 1999
- [49] P. Brössler (SDS GmbH, Vienna), "*Reengineering - Experiences and Some Lessons*", Keynote session at WCRE'2001, Stuttgart, October 2001