Bridging Program Comprehension Tools by Design Navigation

Sébastien Robitaille Reinhard Schauer Rudolf K. Keller
Département IRO
Université de Montréal
C.P. 6128, succursale Centre-ville
Montréal, Québec H3C 3J7, Canada
+1 (514) 343-6782
{robitais, schauer, keller}@iro.umontreal.ca

Abstract

Source code investigation is one of the most time consuming activities during software maintenance and evolution, yet currently available tool support suffers from several shortcomings. Browsing is typically limited to low-level elements, investigation is only supported as a one-way activity, and tools provide little help in getting an encompassing picture of the system under examination. In our research, we have developed tool support for design navigation that addresses these shortcomings. A Design Browser allows for flexible browsing of a system's design level representation and for information exchange with a suite of program comprehension tools. The browser is complemented with a Retriever supporting full-text and structural searching. In this paper, we detail these tools and their integration into a reverse engineering environment, present three case studies, and put them into perspective.

1. Introduction

Understanding source code plays a prominent role during software maintenance and evolution. It is a time consuming activity, especially when dealing with large-scale software systems, and it can rapidly turn into a major bottleneck for software evolution. Zawinski, for instance, attributed one of the main causes for the development slow-down of the *Netscape Communicator* project to the fact that "it takes a long time for a new developer to dive in and start contributing" [29].

Singer et al. define *Just In Time Comprehension* as the source code exploration activity performed by software engineers that leads to software understanding [20]. Software engineers are doing *Just In Time Comprehension* by repeatedly searching for source code artifacts and navigating through their relationships. According to Sim et al., this activity can be split into the two navigation styles normally used for the purpose of information retrieval [19]: *Browsing*, an exploratory and unstructured activity with no specific goal, and *searching*, a planned activity with a specific goal.

Keeping the information retrieval perspective, any collaboration of elements of a software system can be viewed as an *information space*. For example, both physical subsystems, such as files or directories, and logical subsystems, such as class collaborations, are information spaces. These spaces are subsets of the global information space, which represents the whole system. *Information views* are the means to represent information spaces to the software engineer. They normally illustrate information spaces as class diagrams, collaboration diagrams, dependency diagrams, and alike. According to [19], browsing of information views aims at exploring high-level elements of the underlying information space, and searching focuses on retrieving low-level details. Using a combination of these two navigation styles, software engineers can investigate source code and build up a mental model of the system. However, currently available tool support for source code investigation suffers from three major shortcomings.

First, the browsing functionality of current tools is restricted to few high-level elements. However, most high-level elements are of interest during investigation, and hence design level querying and navigation should be supported. Furthermore, browsing should be extensible in the sense that customized queries can be defined. Current tools fail to provide such flexibility.

Second, source code investigation is typically supported as a *one-way trip* from one information view to the source code of the system, which prevents the software engineer from bringing the acquired knowledge back to

This research was supported by the SPOOL project organized by CSER (Consortium for Software Engineering Research) which is funded by Bell Canada, NSERC (National Sciences and Research Council of Canada), and NRC (National Research Council of Canada).

the information view where the navigation started. Thus, proceeding in an *iterative* manner by using the acquired knowledge in subsequent investigations is precluded. Furthermore, as an investigation normally comprises many little steps of browsing and searching, failing to preserve such knowledge may distract the software engineer from the original goal of the investigation and thus add significant overhead to program comprehension [23].

Third, current tools provide little help in getting an encompassing picture of the system under examination. They fail to join the knowledge acquired from different investigations done in different contexts, that is, from browsing different information views. Consider a puzzle where all the pieces put together at the right positions lead to the complete picture. Tools should not only support browsing and searching for software artifacts and identifying the relevant pieces of the puzzle, but they should also allow for the rapid and easy navigation between different views describing the system at different abstraction levels, and thus reduce the time needed to put the pieces of the puzzle to the right place.

In our research, we have developed tool support for design navigation that addresses these shortcomings. A *Design Browser* allows for browsing the source code *model* of a given system, that is, the system's design level representation, and for exchanging information with a suite of program comprehension tools. The browser is complemented with a *Retriever* supporting full-text and structural searching. These two tools allow for bridging the various information views describing the system, and thus joining the different pieces of the puzzle. The tools have been implemented as part of the SPOOL reverse engineering environment. Preliminary experience within our research team and at Bell Canada, our industrial partner, suggests that these tools and the underlying technology substantially ease the task of source code investigation.

Section 2 of this paper describes the SPOOL environment, the testbed in which this research was conducted. In Section 3, the Design Browser and the Retriever are detailed. Section 4 illustrates and assesses these tools, by presenting three case studies where design navigation helped bridging different information views. Section 5 discusses implementation and experience, performance, and related work. Section 6 concludes the paper and provides an outlook into future work.

2. SPOOL environment

The purpose of the SPOOL reverse engineering environment is to help software engineers understand, maintain, and assess software. It has been designed to support software investigation at various levels of abstraction; yet, it emphasizes the design level, in order to complement and bridge architecture and source code level investigation.

The environment has at its core a *repository* that stores *source code models* (see Figure 1). The environment comprises tools to populate the repository, to execute queries on the elements contained in the repository, and to extract and visualize information from these elements. *Source code capturing* can be achieved using *Datrix* [3], *GEN++* [5], or *SNiFF+* [21], and the parsed code is stored in an intermediate, ASCII-based file format. An *importer* is used to parse the *intermediate format* and to provide the repository with the relevant source code elements, that is, the information that is required to support the various levels of source code investigation (for the list of elements contained in the repository, refer to [14]). The schema of the repository is based on the UML metamodel 1.1 [27], which we extended to support the specifics of C++, such as friendship, unions, and enumerations [12]. The object-oriented database management system *Poet 6.0* [15] serves as the repository backend, and the schema is represented as a Java class hierarchy. Given this repository architecture, the elements in the database can be accessed like normal Java objects and used to build graphical representations in form of diagrams (information views).

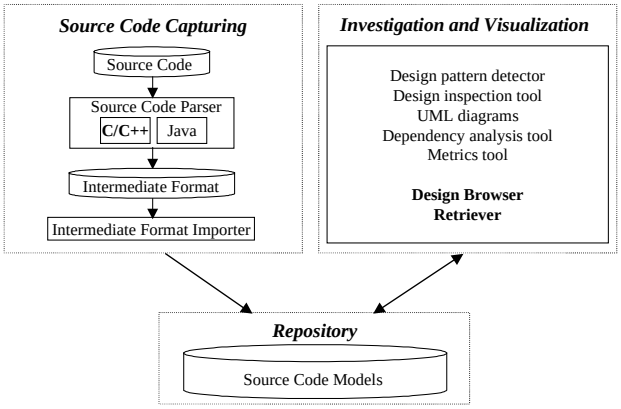

Figure 1: SPOOL architecture

The SPOOL environment provides a number of tools for *investigation and visualization*. At the design level, the *design pattern detector* allows for the identification of implemented design patterns in the system, and the *design inspection tool* supports the visualization of the participants of the recovered patterns within collaboration diagrams [14]. At the architecture level, different *UML diagrams* are provided for the exploration of the system's architecture, and the *dependency analysis tool* allows for the visualization of dependencies between packages, files and classes (such as generalizations, instantiations, operation calls, friendships, and alike). At the source code level, via an integration mechanism enabling

communication between SPOOL and external tools, source code inspection is supported by providing access to the *SNiFF+* source code engineering environment [21]. Furthermore, the *metrics tool* is available for the computation and visualization of metrics. Finally, the *Design Browser* and the *Retriever* provide support for design navigation.

3. Support for design navigation

Design navigation is supported by two tools: the *Design Browser*, which allows for browsing the source code model and exchanging results with the different SPOOL tools, and the *Retriever*, which supports full-text and structural searching. They are described in the sections below.

3.1. SPOOL Design Browser

The user interface of the browser is separated into the three sections *Starting Point*, *Queries*, and *Results* (see Figure 2). The first and the last of these sections may contain each a list of *ModelElements*[1], characterized by their names and by different colored icons that represent their kind (such as *C* for class, *F* for file, and *M* for method). Selecting a *ModelElement* in the *Starting Point* section starts browsing, whereupon the browser displays in the *Queries* section a predefined list of queries for that *ModelElement*. After the query is executed, the result is shown as a new list of *ModelElements* in the *Results* section. For example, Figure 1 shows the browser after the execution of a query that retrieves all the friend classes of *ET_ObjList*, a class from the system *ET++* [9].

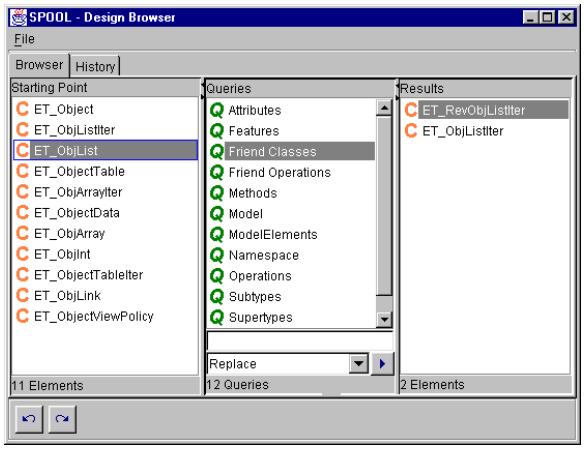

Figure 2: The SPOOL Design Browser

To make querying more flexible and to support bridging the browser with the other tools of the environment, the Design Browser comprises a Drag and Drop (DnD) mechanism. For one thing, this mechanism allows the user to move *ModelElements* back and forth between the Results and the Starting Point sections of the browser, and, in case more than one browser is active, between any pair of browsers. In this way, the result elements of a previous query may serve as the starting point of a new query. Note that there is also an integrated history mechanism that allows the retrieval of previous states of the browser using back and forward buttons (Figure 2). Furthermore, the DnD mechanism makes it possible to move elements between the browser and any other SPOOL tool. In the target of the DnD operation, be it another SPOOL tool or a SPOOL Design Browser, the user is asked via a popup menu to specify the *drop action* that should be performed with the elements being moved.

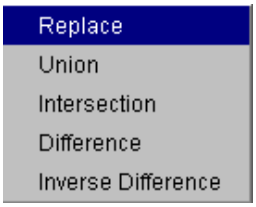

Figure 3: Drop actions in the browser

Possible actions include copy, move, and visualize. In the case of a DnD operation inside a browser or between two browsers, the popup menu shown in Figure 3 appears. The drop actions are *set operations* such as union, intersection, and difference, to be executed on the set of elements being moved and the set of elements in the target section. Note that the entries of both the *Starting Point* and the *Results* sections are internally represented as sets.

The SPOOL Design Browser is query-based. The underlying query mechanism relies on the internal representation of the *ModelElements* of the SPOOL repository. Since the source code model of a parsed system is represented as an object-oriented class hierarchy, many elements for querying are already built-in as Java methods and attributes of the classes that represent the source code elements. As an example, consider *UmlClass*, *UmlAttribute*, and *UmlMethod*, the Java classes in the schema that represent the corresponding UML *ModelElements*. Instances of these classes represent the elements of the system being studied. Suppose *aClass* is an instance of *UmlClass*, and *aMethod* is an instance of *UmlMethod*, then:

- `aClass.getFeatures()` returns the set of all *UmlOperation*, *UmlMethod* and *UmlAttribute* instances contained in that class.
- `aMethod.getCalledOperations()` returns the list of *UmlOperation* that are called by that method.

[1] A *ModelElement*, as described in the UML documentation, is "An element that is an abstraction drawn from the system being modeled" [27]. Accordingly, a *ModelElement* can represent object-oriented concepts such as class or method, as well as files, utilities, and alike.

The predefined queries provided by the SPOOL Design Browser are all based on these and other methods of the SPOOL repository schema.

Browsing exclusively via predefined queries is not sufficient in many cases, and more specific queries may be needed. Therefore, the SPOOL Design Browser allows for the easy addition of new queries which must be written in Java and which must implement a simple predefined interface. Of course, users who want to write custom queries must know about the SPOOL schema. Yet, since the schema follows closely the UML metamodel, this requirement should not be a major obstacle.

The steps to add a new query are simple. As a first step, the user must implement an abstract *execute* method of a predefined *Query* class which is provided by the SPOOL environment. The second step is to add this new query to the browser. The user must provide the query name, its Java package name, and the kind of elements on which the query can be executed, by editing a simple configuration file. Finally, the browser uses Java's reflection mechanisms [10] to instantiate, visualize, and execute the new query.

3.2. SPOOL Retriever

The SPOOL Retriever allows the user to search for a string of characters in the names of the *ModelElements* that are contained in a namespace. A namespace, as described in the UML metamodel documentation, is "a part of the model in which the names may be defined and used. Within a namespace, each name has a unique meaning" [27]. Accordingly, a namespace class can be found in the SPOOL schema, allowing the Retriever to

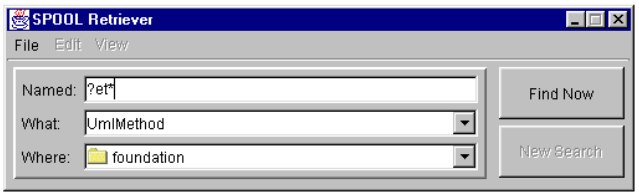

Figure 4: The SPOOL Retriever

search elements that are contained inside other elements such as the whole system, or packages, directories, files, and classes.

Before starting a search, the user specifies in a dialog box (see Figure 4) the search string (*Named*), as well as the kind of element (*What*) and the container (*Where*) of the search. Note that the specification of the search string is optional, and that the Retriever allows the use of wildcards like * and ? to find elements in the system. As an example, Figure 4 shows how *get/set* methods can be retrieved from a particular container (directory, file, class, and alike). This combination of full-text search and structural search is a feature that is typically absent in search tools that are entirely text-based, such as the family of Unix *grep* tools. The structural search capability provided by the SPOOL Retriever is quite powerful and is

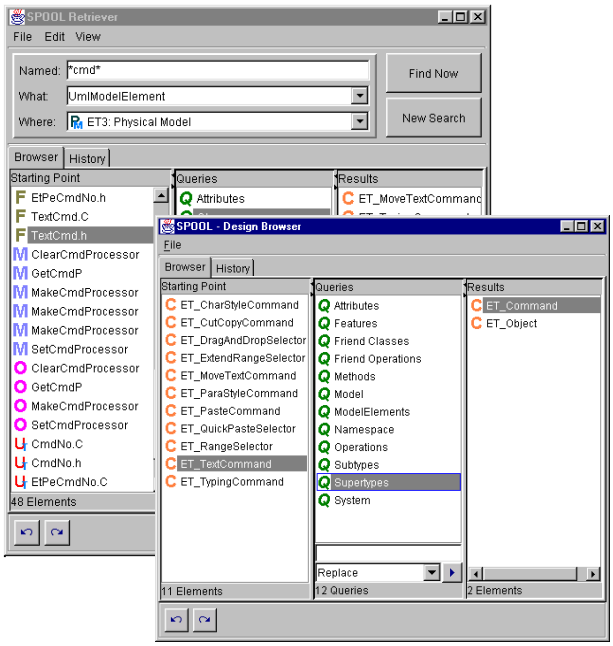

Figure 5: Use of retriever and browsers in concert

due, similarly to the case of the SPOOL Design Browser, to the expressiveness of the schema of the underlying source code models.

The results of a search are displayed in the *SPOOL Design Browser*, to allow for further investigation of the result elements. As an example, in Figure 5 a search for all the *UMLModelElements* in the *ET3:PhysicalModel* that contain the substring *cmd* in their names has been launched, by pressing the *Find Now* button. The results are displayed in the Starting Point section of the top left browser. From there, navigation is continued, by querying about *Classes* and by spawning the button right browser.

The *SPOOL Retriever* can also be applied to recover design concepts, such as design patterns. For instance, in ET++ [9] we found many instances of the *Iterator* [8] pattern just by searching for the expression *iter** in method names. As another example, instances of the *Observer* pattern may be identified by looking for method names containing **observ**, **updat**, or **notif**. Note, however, that for general pattern recovery a more systematic and human-controlled approach is needed [14]. Nevertheless, this simple method can lead to many instances and provide a good starting point for further pattern-based investigations.

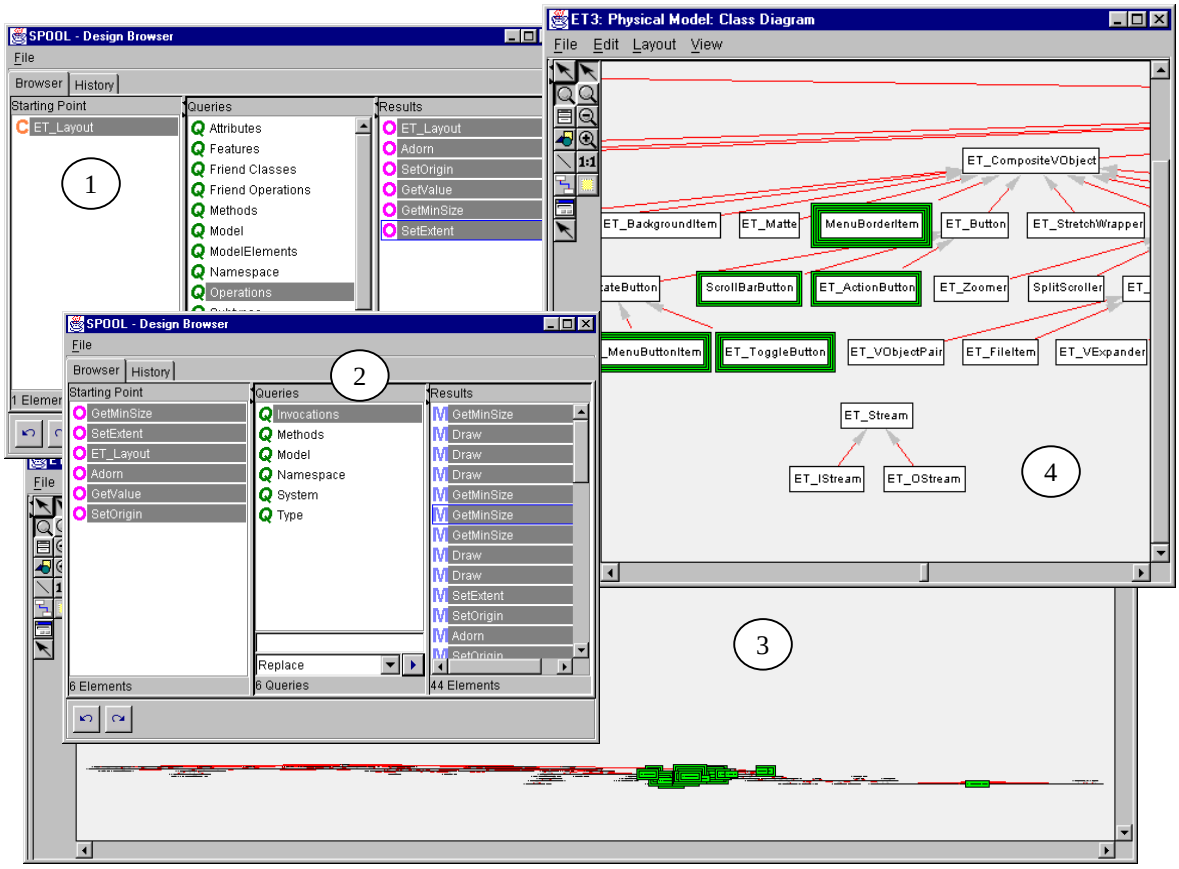

Figure 6: Context investigation and visualization

4. Case studies

In the previous section we described how the SPOOL design navigation tools support browsing and searching the elements and relationships of a system and set the stage for further investigation. In this section, we present three case studies that demonstrate how the SPOOL Design Browser and the SPOOL Retriever can be used to bridge the different investigation tools of the environment and how program comprehension is supported. The case studies address context investigation and visualization, namespace dependency analysis, and design pattern recovery and analysis.

4.1. Context investigation and visualization

The investigation and visualization of contexts is key to program comprehension. This is supported by the tool integration mechanism of the SPOOL environment, which enables tools to communicate without being tightly coupled. The design of SPOOL is event-based, allowing the tools to send events to each other through established event channels [4]. Any SPOOL tool can request from the SPOOL *tool factory* the instantiation of another tool, and send it an event with a set of *ModelElements* as the starting point for further investigation. From the user's point of view, such event-based communication is triggered by using the *drag and drop (DnD)* mechanism described in Section 3.1.

Figure 6 illustrates a typical investigation and visualization scenario. In this example, the user launches the browser to retrieve all the methods that are calling the operations defined in the class *ET_Layout* (windows 1 and 2). Then, the user requests from the tool factory the instantiation of a class diagram and drags the methods into that new diagram. Upon selection of the *visualize* action in the popup menu of the class diagram, these methods (*ModelElements*) are visualized by drawing bounding boxes around the classes in which they are defined (window 3). It is then possible to zoom-in and zoom-out the diagram in order to investigate each of the retrieved classes. For instance, window 4 shows a zoomed part of the cluster of retrieved classes found in the center of window 3. The browser may be used again to go further in the investigation of the system, by starting from one or several classes shown in windows 3 or 4.

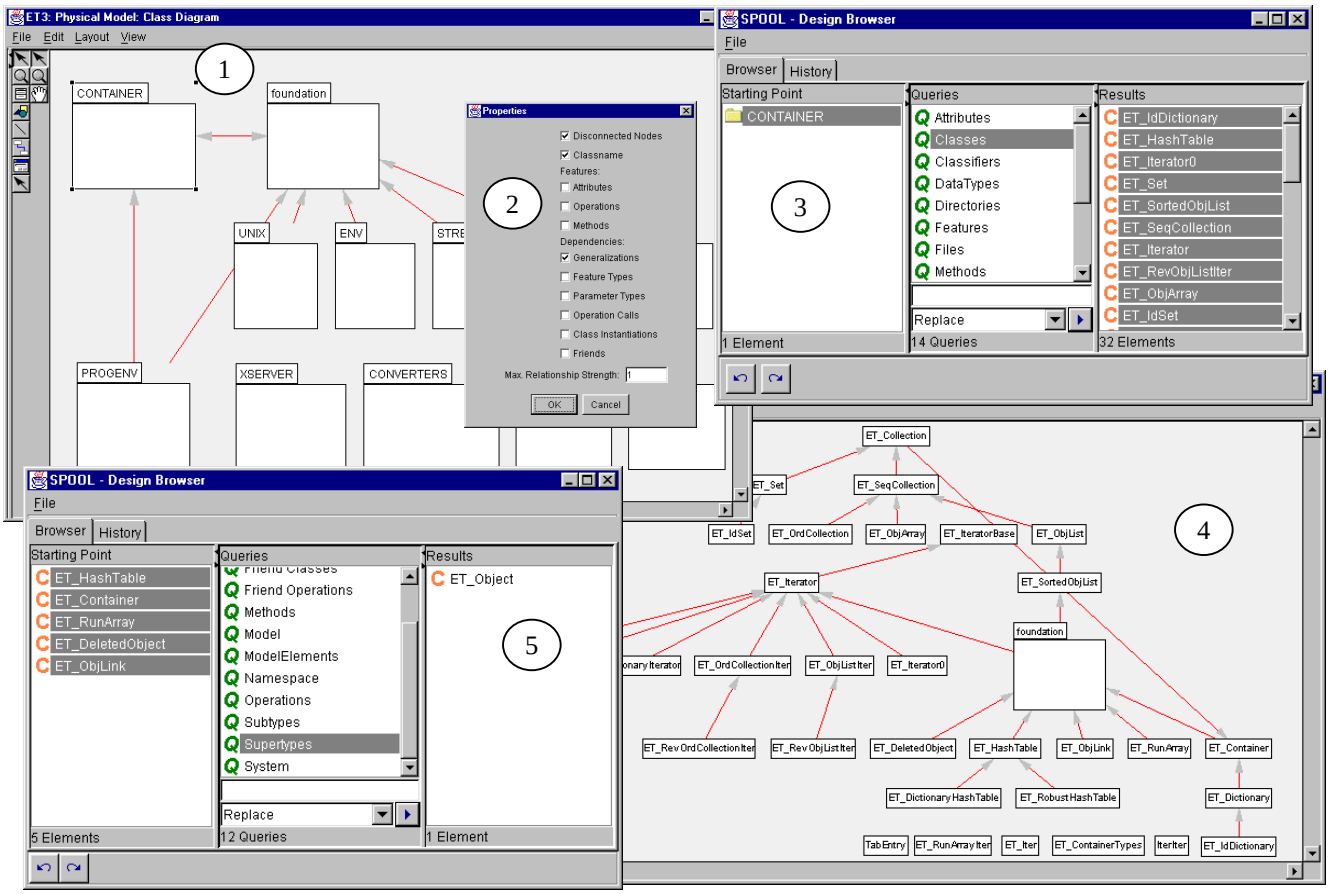

Figure 7: Namespace dependency analysis

4.2. Namespace dependency analysis

Another important activity of program comprehension is the investigation of dependencies among a program's constituent parts. To this end, the SPOOL environment comprises a dependency analysis tool (Figure 7, windows 1 and 4), which visualizes mutual dependencies among namespaces, such as directories, files, classes, and unions. It is quite an intuitive tool that helps understand both desirable and undesirable dependencies among the elements of the system being studied. Particular types of dependencies, such as those based on generalization, operation call, or class instantiation relationships, may be displayed or hidden, according to the settings specified in the diagram's property sheet (Figure 7, window 25). The Design Browser enhances the capabilities of the dependency analysis tool in that it can serve as the data feed as well as the data sink during dependency analysis.

As an example, assume that the user retrieves all directories from ET++ using the Design Browser. Then, he or she selects a few of these directories and drags them over into a dependency analysis diagram (Figure 7, window 1) where they are displayed in form of UML package symbols. The diagram automatically loads the dependencies among these directories from the repository, and visualizes them as uni- or bi-directional lines, respectively. In window 1, only generalization dependencies are shown, according to the specification provided via the property sheet (window 2). Then, the user selects in the dependency analysis diagram the directory *CONTAINER* and invokes a Design Browser from this symbol (window 3). Thereafter, he or she executes a query on that directory to retrieve all the contained classes, and drags the result elements into another dependency analysis tool (window 4) containing initially only the *foundation* directory. By putting these elements together, the user can investigate the bi-directional inheritance relationships between the classes of the *CONTAINER* directory and the classes contained in *foundation*. From the set of five classes that inherit from the classes contained in *foundation*, a further browser is started (window 5), where the user queries the superclasses of the set of classes. The only superclass is *ET_Object*, which is the root class of the foundation package.

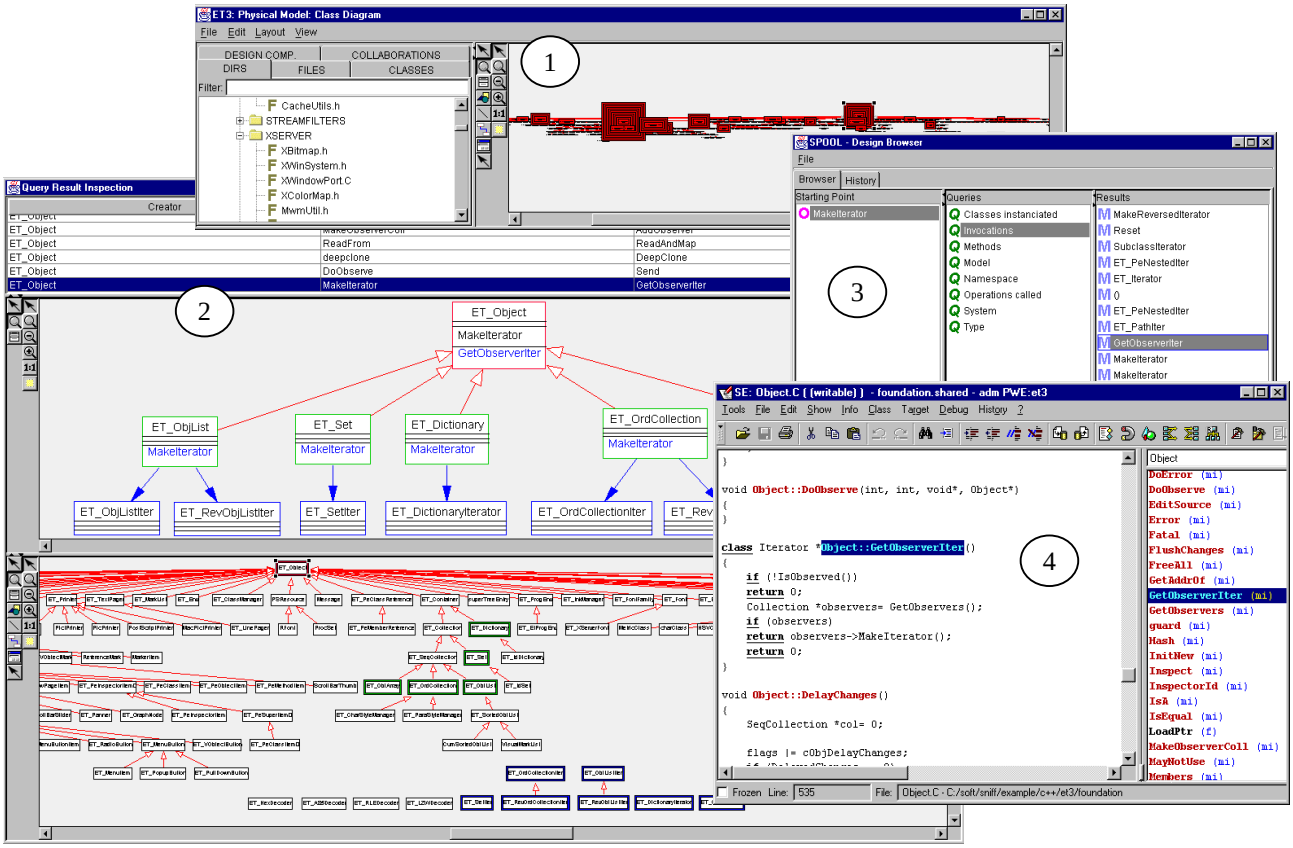

Figure 8: Design pattern recovery and analysis

4.3. Design pattern recovery and analysis

In many object-oriented software systems, the key design decisions are implemented based on well-known design patterns, such as those of Gamma et al. [8], Buschmann et al. [4], and Schmidt [18]. In [14,17], we have reported on automated support for the recovery and analysis of some of these patterns. In this section, we present parts of the functionality implemented to this end, and discuss how design navigation fits in.

As an example, window 1 of Figure 8 illustrates the occurrences of the *Factory Method* pattern [8] in ET++. The recovered pattern instances are visualized by bounding boxes that are incrementally drawn around their so-called reference class, which can be selected by the user. In the example, the reference class chosen by the user is the class that owns the *Creator* role in the *Factory Method* pattern. The size of the outermost bounding box of a class indicates the number of *factory methods* that are implemented in that class. As a second step, the user inspects the recovered pattern instances by starting the design inspection tool (window 2). This diagram shows in its upper part the list of recovered Factory Method patterns, identified by the Creator class and the factory method. The middle part shows the selected Factory Method as a collaboration diagram. The example shows the class *ET_Object* with the method *GetObserverIter* calling the factory method *MakeIterator*[2] (top row). This method is overridden in five subclasses of ET_Object (middle row). Each of these implementations of MakeIterator instantiates different *Products* (bottom row). The lower part of the window shows the recovered classes in the context of the overall class hierarchy. This example presents the case where the design patterns Factory Method and Iterator are combined to provide for a flexible traversal mechanism of ET++ containers, such as lists, sets, dictionaries, collections, and arrays. Recall that the SPOOL Design Retriever might have been used to hint at instances of the Iterator pattern (cf. Section 3.2).

The content of the design inspection diagram was automatically generated by one of the SPOOL design recovery queries for the Factory Method pattern. The diagram provides precious information for program comprehension as it presents in a concise way all the classes that take on some role in a pattern-based

[2] MakeIterator is defined in the class *ET_Container* which is a subclass of ET_Object and a superclass of the classes redefining MakeIterator.

collaboration. Note that in the physical file structure, these classes may be spread out over many directories and subsystems. Yet, the diagram falls short in conveying all the information a user might wish to obtain about the design fragment being studied. He or she might want to know, for instance, the classes and methods that invoke MakeIterator, or get information about the semantics of the method GetObserverIter, whose name alludes to its purpose of creating an Iterator of the Observers of a view element. A visual design inspection tool can never answer all of these questions.

The SPOOL Design Browser together with the SPOOL mechanism for integrating external tools provides the flexibility to obtain detailed knowledge as well as context information about the constituents of a recovered, pattern-based design. For example, the browser of window 3 shows all the methods from which MakeIterator is invoked, including the GetObserverIter method already identified in window 2. By invoking the *SNiFF+* environment, the user can then investigate and edit the retrieved elements directly in the *SNiFF+* source code editor (window 4). This provides invaluable context information about how, in our example, a Factory Method is used.

5. Discussion

In this section, we first report on the implementation and our experience with the SPOOL design navigation tools. Then, performance issues are discussed. Finally, we provide an overview on related work.

5.1. Implementation and experience

The SPOOL Design Browser and Retriever have been implemented in Java, using the *JFC/Swing* components [11]. The implementation consists of some 80 classes and 4,100 lines of code (LOC; comment lines not counted). The complete SPOOL environment comprises currently 640 Java classes and 60,8K LOC; to run the navigation tools stand-alone, the SPOOL framework and repository classes are needed, making up for 240 classes and 17,1 K LOC.

The SPOOL environment, including prototype versions of the navigation tools presented in this paper, has been used extensively as a vehicle for doing joint research with Bell Canada. It played an important role in our research on hot spot recovery [17] and on method replacement inspection and analysis [12]. For example, the capability to display only selected elements of a system in a dependency tool, the possibility to run customized queries on these elements to navigate through their relationships, and the support for visualizing the result elements in known context views were all features that have proven invaluable in the investigation of the large-scale industrial systems that we are considering in our work. Recently, a prototype stand-alone browser has been made available to Bell Canada's quality assessment team, our industrial partner group. Preliminary experience suggests that the browser is indeed useful for the team's daily work.

5.2. Performance

Tool performance is critical for the success of source code investigation. Each step in the investigation process should be fast enough in order to avoid confusion and disorientation with the user. In the following, we present two anecdotal experiments in which the performance of SPOOL queries was measured. The experiments were done on a 350MHz Pentium II machine with 256Mb of RAM running Windows NT 4.0.

The first experiment consisted in measuring the times needed to execute a built-in query, that is, a query that is not directly supported by the metamodel schema. Table 1 shows the data for such a query, that is, the times needed to retrieve all the *ModelElements* (directories, files, classes, C++ structures, C++ unions, C++ enumerations, operations, methods, and attributes), for three industrial C++ software systems. ET++ 3.0 and ACE [25] are both well-known application frameworks. System A is a large-scale system from the telecommunications domain provided by Bell Canada (for confidentiality reasons we cannot disclose the real name of the system).

C++ Systems	# Elements	Duration 1 (seconds)	Duration 2+ (seconds)
ET++	20868	22	2
ACE	21577	49	3
System A	47834	47	6

Table 1: Performance for a built-in query

The table shows that the first time the browser runs a query, it takes longer (*Duration 1*) because, first, Poet needs to recreate the persistent objects that are stored on disk and, second, when loading a system, SPOOL caches some of the objects in internal hash tables. As soon as an element is "touched" by a query, it becomes available in memory, and the next time any query is accessing it, the execution is much faster *(Duration 2+).*

C++ Systems	Total number of template methods found	Duration (seconds)
ET++	371	15
ACE	21	23
System A	364	360

Table 2: Performance for the *Template Method* query

		Discover	Visual Age for Java	SNiFF+	Source-Navigator	SPOOL
Browsing	Built-in queries	+	-	-	-	+
	Customized queries	-	-	-	-	+
	Bridging	-	-	-	-	+
Searching	Full-text	+	+	+	+	+
	Structural	+	+-	+-	+-	+

Table 3: Feature comparison of commercial tools and SPOOL

The above experiment shows that the retrieval of elements that are already referenced in the database is pretty fast. The execution of more complicated (not built-in) queries may take considerably longer. As a second experiment, we measured the time needed to retrieve all occurrences of the Template Method [8] pattern in the three systems. This query basically consists of the following five steps::

1. retrieve all classes in the system,
2. for each class, retrieve all methods,
3. for each method, retrieve all call actions,
4. for each call action, get the receivers,
5. for each receiver, look if the call action is defined in the same class and implemented in a subclass.

Table 2 shows the times needed to execute this query for the first time, assuming that all the *ModelElements* are already cached (a query that retrieves all *ModelElements* in the system was executed previously). These numbers are quite good considering that a very high number of relations must be crossed in order to retrieve the desired information. The time needed to run a particular query may be higher, but these experiments suggest that only the complexities of the query and of the system are susceptible to increase execution time, whereas the access time to the *ModelElements* of the repository is relatively constant (mainly due to the use of hash tables).

5.3. Related work

Many tools have been developed for program comprehension, especially in academia. Some interesting academic tools in respect to browsing and searching include *SHriMP* [24], the *Searchable Bookshelf* [19], and *tksee* [20]. *SHriMP* is a tool that allows navigation of source code using hyperlinks and provides some support for context navigation. Browsing is quite limited in that there are few built-in queries and query customization is not supported. The *Searchable Bookshelf* is a system that provides advanced capabilities for generating and navigating software structure diagrams (called landscapes). It comprises a query language for searching and browsing a system's fact base. The presentation of query results is purely textual, and DnD-style bridging of tools is not supported. *Tksee*, finally, is a representative of the tools that are designed to allow software browsing and searching, but only at the source code level.

On the commercial side, several tools are available that exhibit capabilities for software comprehension. Typically, they provide, beyond these capabilities, support for software development in general. We conducted an informal comparison between the SPOOL navigation tools and four of these tools that we consider most representative for the state-of-the-art, namely, *Discover* [7], *Visual Age for Java* [28], *SNiFF+* [21], and *Source-Navigator* [22]. At the time of the writing of this paper, the comparison is based on practical experience with a professional license of *SNiFF+*, evaluation licenses of *Visual Age for Java* and of *Source-Navigator*, and on the study of the documentation of *Discover* that was made available to us [1,2,26]. A more comprehensive comparison has been incepted and will be described in [16]. The informal comparison considers three browsing features and two searching features (see Table 3): availability of built-in query lists, capability of specifying customized queries, and possibility of bridging browsing with other software comprehension tools as well as support of full-text searching and of structural searching. For each tool, the support of these features was weighted as strong (+), medium (+-), or weak (-).

Table 3 summarizes the results of the comparison. The SPOOL navigation tools, which were especially designed to support the considered features, naturally stand out with strong support of all the features. Three observations are worth mentioning. First, the comparison suggests that the commercial tools have limited browsing capabilities, since except for Discover, they provide little support of built-in querying permitting the navigation of the system relationships and because none of them supports the customization of queries. Second, mechanisms for the

bridging of browsing and other software investigation techniques are not present either (in the case of Discover, no such mechanism is described in the available documentation). We believe that support for these browsing features is of invaluable help for program comprehension, especially if it is integrated into a development environment. Third, full-text searching is well supported by all the tools, whereas structural searching is strongly supported by Discover, with the remaining three commercial tools lacking full support of these capabilities. Recall that the SPOOL Design Browser together with the SPOOL DnD mechanism supports the creation of user-defined views or diagrams, which is an additional important feature that is absent from the commercial tools we considered.

6. Conclusion

Mechanisms for context searching and browsing in a development or maintenance environment are key to help software engineers investigate large-scale software systems. We presented various tools that we consider useful for software comprehension, and we detailed the SPOOL Design Browser and Retriever, two tools developed for the purpose of bridging the other comprehension tools. We showed that it is possible to navigate across various information views, that is, the representations of information spaces, by using the query-based mechanism of the browser and the searching capabilities of the Retriever. The three case studies of this paper demonstrated the usefulness of the tools and the underlying communication mechanism for tool integration. Finally, we presented statistics about the performance of the tools to assess their usability, and we compared some of their features with other well-known, considered "state-of-the-art" development environments.

In the future, we intend to formalize the source code investigation model based on context navigation and bridging, for better synchronization of the knowledge acquired from investigations and the tool-provided information views. We also plan to complement the SPOOL tools with mechanisms allowing for on-the-fly construction of context views, and to extend the Design Browser with "query suites" supporting specific analysis tasks, such as dependency analysis, change impact analysis [5] or component-based reverse engineering [14]. We believe that by using this model and the corresponding tools, software engineers will understand large-scale systems faster and more easily, and will get less lost when browsing such systems. Finally, we wish to assess the usability of such tools. In the first place, we aim to collect further feedback from the quality assessment team at Bell Canada, our industrial partner, concerning their work experience with the tools. Moreover we plan to carry out a controlled experiment about this issue.

Acknowledgement

We would like to thank the following organizations for providing us with licenses of their tools, thus assisting us in the development part of our research: *Bell Canada* for the source code parser *Datrix*, *Lucent Technologies* for their C++ source code analyzer *GEN++* and the layout generators *Dot* and *Neato*, and *TakeFive Software* for their software development environment *SNiFF+*.

References

[1] Barr, J., Product Review "Reengineer/SET", Software Development Magazine, August 1996. On-line at <*http://www.sdmagazine.com/breakrm/products/reviews/s968r2.shtml*>.

[2] Barr, J., Product Review "DISCOVER 3.0", Software Development Magazine, March 1996. On-line at <*http://www.sdmagazine.com/breakrm/products/reviews/s963r2.shtml*>.

[3] Bell Canada. DATRIX abstract semantic graph - reference manual. Montreal, Quebec, Canada. January 1999. Available on request from <*datrix@qc.bell.ca*>.

[4] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M., Pattern-Oriented Software Architecture – A System of Patterns. *John Wiley and Sons*, 1996.

[5] Chaumun, A., Kabaili, H., Keller, R. K. and Lustman, F., A Change Impact Model for Changeability Assessment in Object-Oriented. In *Proceedings of the Third Euromicro Working Conference on Software Maintenance and Reengineering*, pages 130-138, Amsterdam, The Netherlands. March 1999.

[6] Devanbu, P. T. GENOA – a customizable, language and front-end independent code analyzer. In *Proceedings of the 14th International Conference on Software Engineering (ICSE'92)*, pages 307-317. Melbourne, Australia. 1992.

[7] Discover online documentation, Software Emancipation Technology. On-line at <*http://www.setech.com/*>.

[8] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software. *Addison-Wesley*. Menlo Park, CA. 1995.

[9] Gamma, E. and Weinand, A., ET++: A Portable C++ Class Library for a UNIX Environment. Tutorial notes. *OOPSLA '90*. Ottawa, ON, Canada. October 1990.

[10] Java documentation, Sun Microsystems Inc. On-line at <*http://java.sun.com/*>.

[11] Java Foundation Classes (JFC) documentation, Sun Microsystems Inc. On-line at <*http://www.javasoft.com/products/jfc/index.html*>.

[12] Keller, R. K., Knapen, G., Laguë, B., Robitaille, S., Saint-Denis, G., and Schauer, R., The SPOOL design repository: Architecture, schema, and mechanisms. In *Hakan Erdogmus and Oryal Tanir*, editors, *Advances in Software Engineering. Topics in Evolution, Comprehension, and Evaluation.* Springer-Verlag, 2000. 28 pages. To appear.

[13] Keller, R. K., and Schauer, R., Towards a Quantitative Assessment of Method Replacement. In *Proceedings of the Fourth Euromicro Working Conference on Software Maintenance and Reengineering*, pages 141-150, Zurich, Switzerland, February 2000. IEEE.

[14] Keller, R. K., Schauer, R., Robitaille, S., and Pagé, P., Pattern-Based Reverse Engineering of Design Components In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 226-335, Los-Angeles, CA, USA. May 1999.

[15] POET Java ODMG Binding documentation. Poet Software Corporation. San Mateo, CA, USA. On-line at <*http://www.poet.com*>.

[16] Robitaille, S. Tool support for understanding industrial-sized, object-oriented software systems. *Master's thesis*, Université de Montréal, Montreal, Quebec, Canada, April 2000. French title: Support informatique à la compréhension des logiciels orientés objet de taille industrielle.

[17] Schauer R., Robitaille S., Keller, R. K. and Martel, F., Hot Spot Recovery in Object-Oriented Software with Inheritance and Composition Template Methods. In *Proceedings of the International Conference on Software Maintenance (ICSM'99)*, pages 220-229. Oxford, England. August 1999.

[18] Schmidt, D., Design patterns for concurrent, parallel, and distributed systems. On-line at <*http://siesta.cs.wustl.edu/~schmidt/patterns-ace.html*>.

[19] Sim, S. E., Clarke, C. L. A., Holt, R. C. and Cox, A. M., Browsing and Searching Software Architectures. In *Proceedings of the International Conference on Software Maintenance (ICSM'99)*, pages 381-390. Oxford, England, August 1999.

[20] Singer, J., Lethbridge, T., Vinson, N. and Anquetil N., An Examination of Software Engineering Work Practices. In Proceedings of CASCON'97, pages 209-223. Toronto, ON, Canada. 1997.

[21] SNiFF+ documentation set. On-line at <*http://www.takefive.com*>.

[22] Source-Navigator documentation set. On-line at <*http://www.cygnus.com/sn/*>.

[23] Storey, M.-A. D., Fracchia, F. D. and Müller, H. A., Cognitive design elements to support the construction of a mental model during software exploration. Journal of Systems and Software, 44(3):171-185, January 1999.

[24] Storey, M.-A. D. and Müller, H. A., Manipulating and documenting software structures using SHriMP views. In Proceedings of the International Conference on Software Maintenance (ICSM'95), pages 275-284. Opio (Nice), France. October 1995.

[25] Syyid, U., The Adaptative Communication Environment: "ACE". Hughes Network Systems. On-line at <*http://www.cs.wustl.edu/~schmidt/PDF/ACE-tutorial.pdf.gz*>.

[26] Tilley, S. R., Discovering DISCOVER. Technical report CMU/SEI-97-TR-012. Pittsburgh, PA, Oct.1997. On-line at <*http://www.sei.cmu.edu/publications/documents/97.reports/97tr012/97tr012title.htm*>.

[27] UML, Documentation set version 1.1. September 1997. On-line at <*http://www.rational.com*>.

[28] Visual Age for Java online documentation, IBM Corporation. On-line at <*http://www.ibm.com*>.

[29] Zawinsky, J., resignation and postmortem, 1999. On-line at <*http://www.jwz.org/gruntle/nomo.html*>.