Université de Montréal

# Tool Support for Context-Based Comprehension of Large-Scale Software Systems

par

Rui Yin

Département d'Informatique et de Recherche Opérationnelle

Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de
Maître ès sciences (M.Sc)
en informatique

March, 2002

Université de Montréal
Faculté des études supérieurs

Ce mémoire intitulé :

Tool Support for Context-Based Comprehension of Large-Scale Software Systems

présenté par :

Rui Yin

a été évalué par un jury composé des personnes suivantes:

Président-rapporteur: Peter Kropf
Directeur de recherche: Rudolf K. Keller
Membre du jury: François Lustman

Mémoire accepté le : 6 août 2002

# Sommaire

La compréhension du logiciel est une activité qui joue un rôle essentiel et primordial dans le cadre des activités de maintenance et d'évolution des systèmes logiciels de grande taille. Même si plusieurs outils d'aide à la compréhension du logiciel existent, ces outils offrent peu de support pour la compréhension de systèmes orientés objet de grande taille. De plus, ces outils offrent peu de support pour la navigation au niveau conceptuel au sein de grands systèmes.

Dans le cadre de cette recherche, nous avons développé une approche nommée *Visualization in Contexts* (*Visualisation en Contexte)* afin d'aider à la compréhension de systèmes logiciels de taille industrielle. Cette approche est basée sur les concepts du paradigme orienté objet et des patrons de conception. Elle vise à offrir des vues multiples du logiciel, qui présentent des niveaux d'abstraction différents, et permet d'effectuer des références croisées entre ces mêmes vues. L'outil *Context Viewer* (*Visualiseur de Contexte*) fut développé et intégré dans l'environnement SPOOL pour fins de validation de notre approche.

Trois exemples et une évaluation de l'outil, présentés dans ce mémoire, permettent de démontrer la façon par laquelle cette approche facilite la compréhension du logiciel.

***Mots clés*** : patron de conception, système orienté objet, rétroconception logicielle, visualisation, vue de contexte, compréhension du logiciel, outil.

# Abstract

Software comprehension is an activity that plays an essential role in the maintenance and evolution of large-scale software systems. Although various software comprehension tools have been developed, these tools offer little support for the comprehension of large-scale, object-oriented software systems. Moreover, these tools provide little help for navigating at the design level of large-scale systems.

In this research, we have developed an approach, called *Visualization in Contexts,* for helping to comprehend industrial-size, object-oriented software. The approach is based on the concepts of the object-oriented paradigm and of design patterns. It provides various context views of the software, with the views being at various abstraction levels and allowing for cross-referencing. A tool, called *Context Viewer,* was developed and integrated into the SPOOL environment to validate our approach.

The three examples and the tool evaluation presented in this thesis show how our approach facilitates the process of software comprehension.

***Keywords:*** design pattern, object-oriented system, reverse engineering, visualization, context view, software comprehension, tool.

# Table of Content


**SOMMAIRE** ..... **III**

**ABSTRACT** ..... **IV**

**TABLE OF CONTENT** ..... **V**

**LIST OF FIGURE** ..... **VII**

**LIST OF TABLE** ..... **IX**

**LIST OF ABBREVIATIONS** ..... **X**

**ACKNOWLEDGMENT** ..... **XII**

**CHAPTER 1 : INTRODUCTION** ..... **1**

1.1 PROBLEM STATEMENT AND SOLUTION APPROACH ..... 1
1.2 MAJOR CONTRIBUTIONS ..... 2
1.3 THESIS STRUCTURE ..... 2

**CHAPTER 2 : BACKGROUND AND RELATED WORK** ..... **4**

2.1 THEORIES AND CONCEPTS FOR SOFTWARE COMPREHENSION ..... 4
*2.1.1 Cognitive Models* ..... *5*
*2.1.2 Object-Oriented Concepts* ..... *6*
*2.1.3 Design Pattern Concepts* ..... *7*
2.2 TECHNIQUES AND TOOLS FOR SOFTWARE COMPREHENSION ..... 9
*2.2.1 Commercial Tools* ..... *10*
*2.2.2 Academic Tools* ..... *11*
2.3 SUMMARY ..... 12

**CHAPTER 3 : THE SPOOL ENVIRONMENT** ..... **14**

3.1 OVERVIEW OF THE SPOOL ENVIRONMENT ..... 14
3.2 THE SPOOL REPOSITORY ..... 17
3.3 THE SPOOL ANALYSIS TOOLS ..... 20
*3.3.1 Analysis at the Source Code Level* ..... *21*
*3.3.2 Analyses at the Structure Level* ..... *21*
*3.3.3 Analyses at the Design Level* ..... *23*
3.4 DESIGN NAVIGATION ..... 25

**CHAPTER 4 : OVERVIEW OF THE *CONTEXT VIEWER*** ..... **28**

4.1 OUR APPROACH: *VISUALIZATION IN CONTEXTS* ..... 28

4.2 REQUIREMENTS FOR THE *CONTEXT VIEWER* ..... 29
4.3 PRINCIPLES OF FEATURE DESIGN ..... 30
4.4 MAIN FUNCTIONS OF THE *CONTEXT VIEWER* ..... 31
*4.4.1 E-Set Elements and Operations ..... 31*
*4.4.2 Various Context Views ..... 31*
*4.4.3 Mechanisms Embedded inside the Context Viewer ..... 40*

**CHAPTER 5 : DESIGN AND IMPLEMENTATION OF THE *CONTEXT VIEWER* ..... 43**

5.1 FACTORS AFFECTING THE DESIGN ..... 43
5.2 CONTEXT VIEWER DESIGN COMPONENTS ..... 44
*5.2.1 ESet Observer ..... 44*
*5.2.2 Text Search Strategy ..... 46*
*5.2.3 Context Viewer Factory ..... 48*
*5.2.4 Event Handler ..... 49*
5.3 CHOICE OF LAYOUT STRATEGY TOOLS ..... 51
5.4 IMPLEMENTATION AND EXPERIENCES ..... 53

**CHAPTER 6 : EXAMPLES ..... 55**

EXAMPLE 1 : INVESTIGATION OF CONTEXTS ..... 55
EXAMPLE 2 : REVERSE ENGINEERING OF DESIGN PATTERNS ..... 57
EXAMPLE 3 : REDUCTION OF COGNITIVE OVERHEAD ..... 59

**CHAPTER 7 : DISCUSSION ..... 61**

7.1 COMPREHENSION APPROACH ..... 61
7.2 EVALUATION OF SPOOL ENVIRONMENT ..... 62
7.3 RELATED WORK ..... 65
7.4 LIMITATIONS ..... 68

**CHAPTER 8 : CONCLUSION ..... 69**

8.1 SUMMARY ..... 69
8.2 FUTURE WORK ..... 70

**REFERENCES ..... 72**

**APPENDIX : SPECIFICATION OF CONTEXT VIEWER ..... I**

# List of Figure

FIGURE 1: ARCHITECTURE OF THE SPOOL ENVIRONMENT ..... 15
FIGURE 2: SPOOL REPOSITORY SCHEMA: CORE CLASSES ..... 17
FIGURE 3: SPOOL REPOSITORY SCHEMA: FEATURE CLASSES ..... 19
FIGURE 4: SPOOL REPOSITORY SCHEMA: ACTION CLASSES ..... 20
FIGURE 5: PROPERTY SHEET IN UML DIAGRAMS ..... 21
FIGURE 6: INHERITANCE DEPENDENCY DIAGRAM ..... 22
FIGURE 7: HIGHER LEVEL DEPENDENCY DIAGRAM ..... 22
FIGURE 8: DESIGN PATTERN DETECTION ..... 24
FIGURE 9: DESIGN INSPECTOR ..... 25
FIGURE 10: DESIGN BROWSER ..... 26
FIGURE 11: RETRIEVER ..... 26
FIGURE 12: SOURCE CODE CONTEXT VIEW ..... 32
FIGURE 13: CONTAINMENT CONTEXT VIEW ..... 33
FIGURE 14: INHERITANCE CONTEXT VIEW ..... 34
FIGURE 15: STRUCTURE OF FACTORY METHOD DESIGN PATTERN ..... 36
FIGURE 16: FACTORY METHOD CONTEXT VIEW ..... 37
FIGURE 17: TEMPLATE METHOD CONTEXT VIEW ..... 38
FIGURE 18: MULTIPLE CONTEXT VIEW ..... 39
FIGURE 19: SYNCHRONIZATION MECHANISM ..... 40
FIGURE 20: HISTORY MECHANISM ..... 41
FIGURE 21 : ESET OBSERVER ..... 46
FIGURE 22: TEXT SEARCH STRATEGY ..... 47
FIGURE 23: CONTEXT VIEWER FACTORY ..... 49

FIGURE 24: EVENT HANDLER ..... 50
FIGURE 25: INVESTIGATION OF CONTEXTS ..... 56
FIGURE 26: REVERSE ENGINEERING OF DESIGN PATTERNS ..... 57
FIGURE 27: REDUCTION OF COGNITIVE OVERHEAD ..... 59

# List of Table

TABLE 1: INFORMATION CONTAINED IN THE SPOOL REPOSITORY ..... 16
TABLE 2: FEATURE COMPARISON OF COMMERCIAL TOOLS AND SPOOL ..... 66
TABLE 3: FEATURE COMPARISON OF ACADEMICAL TOOLS AND SPOOL ..... 67

# List of Abbreviations

| | |
|---|---|
| CSER | Consortium for Software Engineering Research |
| DnD | Drag & Drop |
| ICSE | International Conference on Software Engineering |
| ICSM | International Conference on Software Maintenance |
| NRC | National Research Council of Canada |
| NSERC | National Sciences and Research Council of Canada |
| OSI | Open Systems Interconnection |
| SPOOL | Spreading Desirable Properties into the Design of Object-Oriented, Large-Scale Software Systems |
| TCP/IP | Transfer Control Protocol / Internet Protocol |
| UML | Unified Modeling Language |

I dedicate this thesis

to

my parents, Jing Yu and Bang Xin

# Acknowledgment

I would like to thank Mr. Rudolf K Keller, professor at Université de Montréal, for allowing me to be a member of the SPOOL project group and for having directed and supervised the writing of this thesis. His patience, critical spirit, and great encouragement have provided a priceless assistance throughout this work.

Moreover, I want to thank my colleagues in the GÉLO (GÉnie LOgiciel) group for their support, understanding, and patience.

The SPOOL project is organized by CSER (Consortium for Software Research Engineering), which is financed by Bell Canada, NSERC (Natural Sciences and Engineering Research Council of Canada), and NRC (National Research Council Canada). I would like to thank the members of these organizations who helped making the realization of this research possible.

Lastly, I would like to thank my parents who have always been supportive, especially during the difficult phases of this work.

# Chapter 1 : Introduction

Software comprehension is an activity that plays an essential and dominating role during the maintenance and the evolution of software systems. It is an activity that can take programmers much time, especially when the system under investigation is large-scale (typically with several millions lines of code) and when the programmers are new to the software system.

## 1.1 Problem Statement and Solution Approach

Bell Canada spends millions of dollars each year for purchasing and maintaining large-scale software systems written in C++. In order to maintain and add new functionality into these systems in the future, Bell Canada wants to assure that the software systems under investigation are of a certain quality. To this end, there is a quality assurance team at Bell Canada, which uses several tools and techniques to assess software systems before their initial purchase, during their development and all phases of their evolution. This team needs reverse engineering tools to help understand the original design of the system and the design decisions taken by its programmers. Even though various software comprehension tools exist, they offer little support in the comprehension of large-scale, object-oriented software systems. They take little advantage of the additional information available in the source code of object-oriented systems. Moreover, these tools provide little help for the design-level navigation of large-scale software systems.

The work presented in this thesis was conducted in the project SPOOL (Spreading Desirable Properties into the Design of object-oriented, Large-Scale Software Systems), a CSER (Consortium for Software Engineering Research) project that is carried out as a collaboration between Université de Montréal and Bell Canada. The main interest of this

project is to identify the desirable properties in the design of large-scale, object-oriented systems, and to be able to evaluate their quality in respect to those properties. This thesis presents a new approach, namely *Visualization in Contexts,* to better support the comprehension of large-scale, object-oriented industrial software systems as well as a prototype tool, namely the *Context Viewer,* which supports this comprehension approach within the SPOOL environment. The initial idea for the *Context Viewer* is discussed in [19], which is the starting point for our research.

## 1.2 Major Contributions

This work makes two major contributions. The first contribution is the software comprehension approach *Visualization in Contexts,* together with the two concepts view synchronization and design pattern view. *Visualization in Contexts* is based on the utilization of the object-oriented paradigm and the design patterns concept, which are hardly found so far in other work in the reverse engineering domain. Our research thus opens up new possibilities in terms of how to facilitate the software comprehension process.

The second major contribution of this work consists in the *Context Viewer*, which is a tool that implements the *Visualization in Contexts* approach and related concepts. The *Context Viewer* is integrated in the SPOOL environment. Consequently, it is available for installation at Bell Canada and evaluation in an industrial context.

These contributions are summarized in the paper *Program Comprehension by Visualization in Context* by Rui Yin and Rudolf K. Keller. The paper has been accepted as a technical paper at the upcoming *International Conference on Software Maintenance (ICSM'2002)*, Montreal, Canada, October 2002 [21] (rigorously refereed conference with acceptance rate below 50%).

## 1.3 Thesis Structure

Chapter 2 presents various theoretical aspects related to software comprehension. It discusses several cognitive models, as well as various comprehension concepts related to the object-oriented paradigm and design patterns. This chapter also gives an overview of existing commercial and academic software comprehension tools.

Chapter 3 reviews the SPOOL reverse engineering environment, including its design repository and its tools for analysis at various levels of abstraction.

Chapter 4 gives an overview of the *Context Viewer*; and describes the approach *Visualization in Contexts,* which is at the core of the *Context Viewer.* It explains the requirements of the *Context Viewer* and reviews the main functionality of the tool.

Chapter 5 discusses the design and implementation aspects of the *Context Viewer.* It presents various factors that affect the design of the tool as well as some of the design components employed in its implementation. It also details the considerations about the selection of layout strategy tools.

Chapter 6 presents three examples, illustrating the interaction between the *Context Viewer* and other SPOOL tools in facilitating the software comprehension process and showing how the *Visualization in Contexts* approach is supported by SPOOL.

Chapter 7 reports on the evaluation of the context-based SPOOL tools with respect to two requirement lists proposed for program comprehension tools. It also informally compares the SPOOL tools with related commercial and academic tools. Moreover, it summarizes the limits of the SPOOL tools.

Lastly, Chapter 8 briefly summarizes this work and the major contributions. A discussion of future work concludes the thesis.

The Appendix presents the design specification of the *Context Viewer.* It gives a perspective of the *Context Viewer* from the point of view of the developer.

# Chapter 2 : Background and Related Work

During the life cycle of a software system, 60-80 percent of its cost is spent on maintenance and updating. One main reason is that understanding a software system is a difficult activity. It is much more difficult if the software under investigation is of industrial size since this type of software usually contains millions of lines of code and is written by different groups of developers in different styles. Moreover, it involves application domains with which we are not familiar. Reverse engineering has been heralded as one of the most promising technologies to cope with this situation. Reverse engineering research has produced a number of theories for software understanding over the past ten years. In Section 2.1, we will describe some of these theories. In Section 2.2, we will introduce various reverse engineering tools available in both the commercial and academic field and discuss their limitations in supporting the comprehension of industrial-size object-oriented software systems.

## 2.1 Theories and Concepts for Software Comprehension

There are several cognitive models in the reverse engineering domain; they try to solve the problem of software comprehension from different angles, such as top-down or bottom-up. Moreover, the process of building these cognitive models are in general influenced by many factors. These factors could be of a cognitive nature or be related to the approach of comprehension itself. The programming paradigm and the language used could be another factor, and so are the style of programming and design of the software under investigation. In this section, we will analyze some of these aspects in more detail.

### 2.1.1 Cognitive Models

The bottom-up model [26] of comprehension suggests that understanding is built from bottom up, by reading the source code and then mentally chunking or grouping these statements into higher-level abstractions, which in turn will be aggregated into the high-level understanding of the program.

The top-down model, which is the opposite of bottom-up, was introduced by Brooks [2]. According to him, the mental model of the programmer is built in the manner of top-down. This model starts by creating beforehand a hierarchy of hypotheses about the source code. Then, the initial hypotheses are refined by verification and by forming subsidiary hypotheses.

In his knowledge-based understanding model, Letovsky [16] views programmers as opportunistic processors capable of exploiting either bottom-up or top-down cues. The assimilation process in the model describes how the mental model evolves using the programmer's knowledge base together with the source code and the documentation of the system.

The model of Soloway *et al.* [30] merged the concepts of systematic strategies, as-needed strategies and inquiry episodes into a single model by using the concepts of micro-strategy and macro-strategy. The former are used to get an understanding at the local level; the latter are used to achieve an understanding at a more global level.

Von Mayrhauser and Vans [43][44] created a comprehension metamodel, which integrates several aspects of the models described above. This metamodel is made up of four components. The first three components describe the comprehension processes used to create mental representations at various levels of abstraction. The fourth component describes the knowledge base needed for the construction of the three preceding processes. Comprehension is thus done, according to Von Mayrhauser and Vans, by using each one of these processes according to needs (i.e. at the time considered to be convenient by the programmer).

There are disparities in these cognitive models which are in part due to the characteristics of the maintainer, the program to be understood and the goal for comprehending the program. Starting from these, Storey [33] extracted a list of elements that influence the comprehension strategies of the software maintainers. This list is divided into three types of elements: maintainer characteristics, program characteristics, and task characteristics. Starting from these various characteristics, Storey[34] worked out and organized a hierarchy of cognitive design elements to guide the development of tools to aid in the exploration and comprehension of software. This hierarchy is separated into two large branches. The first branch is intended to capture the essential processes of the various comprehension strategies such as the top-down, bottom-up and integrated approaches. The other branch addresses the cognitive issues of the maintainer while he or she browses and navigates the visualization of the software structure. This tree of cognitive design elements thus gives a base for the specification of various software comprehension tools.

### 2.1.2 Object-Oriented Concepts

In the preceding section, we described some cognitive aspects of software comprehension; the models presented above are strongly related to the concept of abstraction. In fact, the ultimate goal of reverse engineering is to generate a mental model about the software system under investigation, which is a rather high-level abstraction in terms of the software system. Object-oriented languages support abstraction at a certain level by the language itself. In this section, we will describe in detail how this will facilitate the comprehension of programs written in a certain language.

During the last two decades, many programming languages were invented and evolved. They have different strengths and are suitable for different types of programming, such as Prolog for logical programming, Lisp for functional programming, Pascal for imperative programming, and C++ and Java for object-oriented programming. The underlying concepts of a programming language introduce certain forms of semantics into the

software. For example, logical programming introduces the concept of predicate; object-oriented programming introduces the concept of object, etc.

Consequently, we can find this semantic information in the source code directly. This kind of programming language related information is essential to the comprehension of the system written in such a programming language. The understanding of a program written in a low-level language like Assembler, for example, can be a very difficult activity. Not only because the programmer has to know and understand many details relating to hardware (registers, memory, etc), but also because the significant abstractions established in the program are not reflected directly in the source code. If the information of the various semantic elements could be extracted from the source code directly, understanding the program would require the programmer much less effort. This is the case with programs that are written in object-oriented programming languages.

The concepts of abstraction and encapsulation are directly supported by the object-oriented programming languages. For example, the concepts of classes and objects present in the various object-oriented programming languages make it very easy to extract different modules of a software system (classes, for example) directly from the source code. Thus, a system can easily be divided into various modules, which can be understood independently. Moreover, inheritance enables abstraction by providing the concepts of generalization and specializations. Polymorphism also brings certain semantics to the programs in that it allows the specification of a method in the program to have several implementations. By taking advantage of these abstract concepts, the comprehension of an object-oriented software system will be largely facilitated.

### 2.1.3 Design Pattern Concepts

To move to a higher level of abstraction during the process of software comprehension, design patterns are a very helpful concept, which can facilitate the comprehension process significantly. In this section, we will introduce the concept of design pattern, then briefly go through different types of design patterns and the reasons why they are so helpful.

Designing object-oriented software is difficult, and designing reusable object-oriented software is even more difficult. Yet, experienced object-oriented designers do make good designs; one reason is that they have knowledge of certain solutions to the current problems and an aptitude to apply these solutions each time this kind of problem arises again. A design pattern can be seen as the description of one of these design problems together with its solutions. In fact, there are several types of design patterns at various levels of abstraction. Buschmann *et al.* [3] make the distinction between architectural patterns, Design Patterns and programming idioms. We will present the definitions given by Buschman *et al.* to each of these types.

Architectural patterns are models for concrete software architectures and are thus patterns at a high abstraction level. They specify the responsibilities of various subsystems and different components in a system, and the rules to organize the relations between them. Buschmann *et al.* introduce several architectural patterns. *Layers* [3] is an architectural pattern that helps to structure applications in different abstraction layers; networking protocols are probably the best-known example of layered architectures. *Reflexion*, *Broker,* and *Pipes and Filters* are other well-known architectural patterns described in [3].

As defined by Gamma et al. [9], a Design Pattern describes a communication structure between software elements and solves a recurring design problem. This type of pattern is best known and there are hundreds of them in different catalogues [3][9][25], concerning very varied domains. *Observer, Bridge, Iterator, State, Strateg, and Visitor*[9] are among the best-known Design Patterns.

Programming idioms are low abstraction level patterns, which are very language-specific. A single idiom might help one to solve a recurring problem with the programming language one is using. Examples of such problems are memory management, object creation, naming of methods, efficient use of specific library components and so on. The *Counted Pointer* [25] is a well-known idiom, which makes the memory management of dynamically allocated shared objects in C++ easier.

During the software comprehension process, the identification of these design solutions will be very helpful and largely facilitates the process. The reason is that the implementation of a design pattern usually involves more than one object; the understanding of the interdependency and collaboration among these objects can thus form a much higher level of abstraction than the abstraction represented by a single class or object.

## 2.2 Techniques and Tools for Software Comprehension

Techniques used to aid program understanding can be grouped into three categories: unaided browsing, leveraging corporate knowledge and experience, and computer-aided techniques like reverse engineering.

For unaided browsing, the software engineer manually flips through the source code. This approach has the limitations that a software engineer may only be able to keep track of a small amount of information in his or her head.

Leveraging corporate knowledge and experience can be accomplished through mentoring or by conducting informal interviews with the personnel knowledgeable about the subject system. This approach can be very valuable if there are people available who have been associated with the system as it has evolved over time. They carry important information about design decisions, major changes over time, and troublesome subsystems. However, leveraging corporate knowledge and experience is not always possible, because in many cases, the software system may have been acquired from another company.

A reverse engineering environment can manage the complexities of program understanding by helping the software engineer extract high-level information from low-level artifacts, such as the source code. This frees software engineers from tedious, manual, and error-prone tasks such as code reading, searching, and pattern matching by inspection. Without reverse engineering tools, it is very difficult to understand large-scale software systems. These kinds of systems often contain several millions lines of code distributed over several thousand files, and the documentation is seldom up to date, often

even non-existent. In the following sections, we will briefly discuss some representative reverse engineering tools found in both commercial and academic environments.

### 2.2.1 Commercial Tools

The commercial tools developed specifically for supporting software comprehension are quite rare. In this section, we will first review *Understand for C++* [39], which is a reverse engineering, documentation and metrics tool for C and C++ source code. Then, we will describe four development environments that have functionalities contributing to software comprehension. Comprehension tools are integrated into these software development environments, so they support continuous program understanding and try to address information needs throughout the software lifecycle.

*Understand for C++* [39] is marketed by the company *Scientific Toolworks, Inc.* It offers code navigation using a detailed cross reference, a syntax colorizing "smart" editor, and a variety of graphical reverse engineering views at the structure, hierarchy, and source code level. The user can analyze various dependencies among entities in the database of the tool. It is designed to help maintain and understand large amounts of legacy or newly created C and C++ source code.

Discover [7] is a development environment, which is marketed by the company *Software Emancipation Technology*. According to available documentation [7][36], it provides a whole range of support tools for the different people involved in the development of a software system. The environment is organized into several tool sets: For example, one of these sets is intended for project managers, another one comprises tools supporting the system architects, etc. The set that is most relevant for this work is the one intended for developers. It comprises a change impact analysis module and a tool called *Visual Navigator* to navigate and make searches in a system by carrying out preset queries. The source code of software systems is stored in the environment database and source code level navigation is supported.

Another development environment, *Visual Age for Java* [41], is marketed *by IBM Corp.* and integrates various functionalities such as the automatic management of versions. The source code of the systems being developed within this environment is managed in the database of the environment. For software comprehension, this environment provides search tools and diagram viewers to visualize the class hierarchy, the method calls, etc.

Lastly, the environment *SNiFF+* [29], a development environment from *WindRiver Systems Inc.*, as well as the *Source Navigator* [31] of *RedHat.com,* are two development environments providing the functionality to enhance software comprehension. Element searching, a cross-reference viewer and a class diagram viewer are available in each of the two environments.

There are several other commercial environments in the market. Yet, we feel that the environments mentioned above are quite representative.

### 2.2.2 Academic Tools

There are a great number of academic tools that were developed with the aim to aid and facilitate software comprehension. They address several aspects of software system comprehension for various programming languages and paradigms. In the following, we will briefly go through four of them that we feel are typical of the state-of-the-art of academic tools.

*ShriMP* [33][34][35] supports system navigation by using hyperlinks inserted directly in the source code. Moreover, source code and documentation are presented by embedding code and document fragments within the nodes of the nested graphs representing the system. *SHriMP* combines this hypertext metaphor with animated panning and zooming motions over the nested graph to provide continuous orientation and contextual cues for the user.

*The Portable Bookshelf* [8][12][27] is an academic environment, which was designed with the aim of providing a standardized format of system documentation. The information on a system is made available in the form of a Web page and can be browsed

by a navigator like Netscape. Users are able to navigate within the interior of diagrams (called Landscapes) that will display the various subsystems and files of the system graphically, as well as the dependencies among them, like the calls of functions, for example. The internal navigation of such diagrams is rather interactive since they are displayed in the navigator by a Java applet.

*Rigi* [45] is a tool, which allows the visualization of diagrams representing the various aspects of a software system (subsystems, files, etc.) in an abstract way. However, in contrast to the preceding tools, it does not provide a search mechanism and thus is less fit for the interactive exploration of a system.

*TkSee* [28] is another tool that can be used for software comprehension. It provides mechanisms of integrated search and navigation and allows the displaying of source codes.

## 2.3 Summary

Software comprehension is a central activity in a variety of maintenance tasks and even in the development process of large-scale software systems, since programmers in different groups need to understand the program written by others. The goal of software comprehension is to derive the abstract representations of a software system from its source code in order to build mental models of the system. In the preceding sections, we reviewed cognitive models of software comprehension and certain aspects of the object-oriented paradigm and of design patterns that help the comprehension process. Then, we examined several tools developed with the aim of supporting software comprehension. Among existing tools, few of them focus on the comprehension of object-oriented software systems. Neither do they take advantage of some useful information in this type of software systems, such as polymorphism, for example. In addition, none of them provides any navigation support at the design level of the software system at hand.

Moreover, most tools available offer little support to our new software comprehension approach, namely *Visualization in Contexts*. We will introduce this approach and

describe the functions demanded by it in Chapter 4, and then conduct the comparison of selected tools with respect to their support to this approach in Chapter 7. In the next Chapter, we will introduce the SPOOL environment, which offers powerful support to the comprehension of large-scale object-oriented systems in both source code navigation and design recovery. The SPOOL environment is designed to help Bell Canada in assessing large-scale software systems written in C++.

# Chapter 3 : The SPOOL Environment

In the SPOOL project (Spreading Desirable Properties into the Design of object-oriented, Large-Scale Software Systems), a joint industry/university collaboration between the software quality assessment team of Bell Canada and the GELO group at the University of Montreal, the SPOOL environment was developed for design pattern engineering. In this chapter (parts of the chapter are based on [23][19]), we first give, in Section 3.1, an overview of the SPOOL environment, and then review its repository schema in Section 3.2. In Sections 3.3 and 3.4, analysis tools (except for the *Context Viewer,* which will be described in the Chapter 4) and design navigation are described, respectively.

## 3.1 Overview of the SPOOL Environment

The SPOOL environment is entirely written in the Java language and makes it possible to store and visualize the static information extracted from the source code of object-oriented software (C++, Java). It was conceived mainly to allow the investigation of large-scale systems at the design level, but it is also capable of exploring such systems at less abstract levels, such as the structure level, or directly at the source code level.

The SPOOL environment is made of several components as shown in Figure 1. The core of the environment is an object-oriented database [18], which is used as a repository for the whole environment and contains the information of the systems to be analyzed. The repository is accessed by a series of visualization tools, such as the UML diagrams (a tool to generate UML style diagrams), the dependency analyzer, the design pattern detector, the design patterns inspector, the search and navigation tool and the metrics tool. The information contained in the repository is imported by using various syntactic analysis techniques. For example, it is possible to use parsers, such as Datrix [1], Discover [7],

and GEN++ [6] to extract information from the source code, and subsequently import it into the repository (Table 1 shows the list of types of information extracted and imported to the SPOOL repository). Other technologies like the environment SNiFF+ [29] also make it possible to extract certain information and import it into the SPOOL repository.

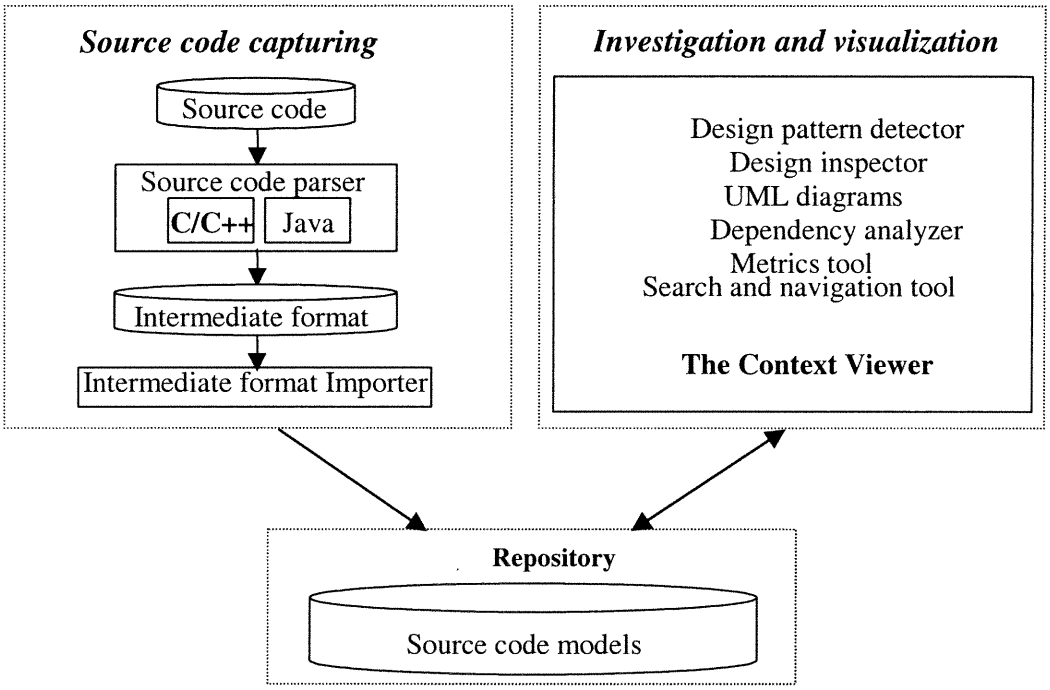


**Figure 1: Architecture of the SPOOL environment**

In fact, the information extracted from the source code is stored in the repository in the form of an object model. This means that each "element" of the source code is represented in the SPOOL environment as a standard Java object, with its type, its attributes and its methods. For example, the user can retrieve not only the classes, the files, the attributes and the methods of the systems at hand in the form of objects from the repository, but also the various types of relations between these elements like the operation calls, the references, instantiations, etc. As all these objects are stored in the database, various tools can thus reach them and navigate through them simply along their interrelationships. This approach allows the using of information contained in the repository inside any Java program in a transparent way. The various SPOOL analysis

tools enumerated previously use this approach and make it possible to visualize, seek and navigate easily the various system elements.

| | |
|---|---|
| 1. | *Files* (name, directory). |
| 2. | *Classifier – classes, structures, unions, anonymous unions, primitive types* (char, int, float, etc.), *enumerations* [name, file, visibility]. Class declarations are resolved to point to their definitions. |
| 3. | *Generalization* relationships [superclass, subclass, visibility]. |
| 4. | *Attributes* [name, type, owner, visibility]. Global and static variables are stored in utility classes (as suggested by the UML), one associated to each file. Variable declarations resolved to point to their definitions. |
| 5. | *Operations* and *methods* [name, visibility, polymorphic, kind]. Methods are the implementations of operations. Free functions and operators are stored in *utility* classes (as suggested by the UML), one associated to each file. *Kind* stands for *constructor*, *destructor*, *standard*, or *operator*. |
| 6. | *Parameters* [name, type]. The type is a *classifier*. |
| 7. | *Return types* [name, type]. The type is a *classifier*. |
| 8. | *Call actions* [operation, sender, receiver]. The receiver points to the class to which a request (operation) is sent. The sender is the classifier that owns the method of the call action. |
| 9. | *Create actions*. These represent object instantiations. |
| 10. | *Variable use* within a method. This set contains all member attributes, parameters, and local attributes used by the method. |
| 11. | *Friendship relationships* between classes and operations. |
| 12. | *Class and function template instantiations*. These are stored as normal *classes* and as *operations* and *methods*, respectively. |

**Table 1: Information contained in the SPOOL repository**

## 3.2 The SPOOL Repository

The schema of the SPOOL repository [23] is an object-oriented class hierarchy whose core structure is adopted from the UML metamodel. Being a metamodel for software analysis and design, UML provides a well-thought foundation for SPOOL as a design comprehension environment. However, SPOOL reverse engineering starts with source code from which design information should be derived. This necessitates extensions to the UML metamodel in order to cover the programming language level as far as it is relevant for design recovery and analysis.

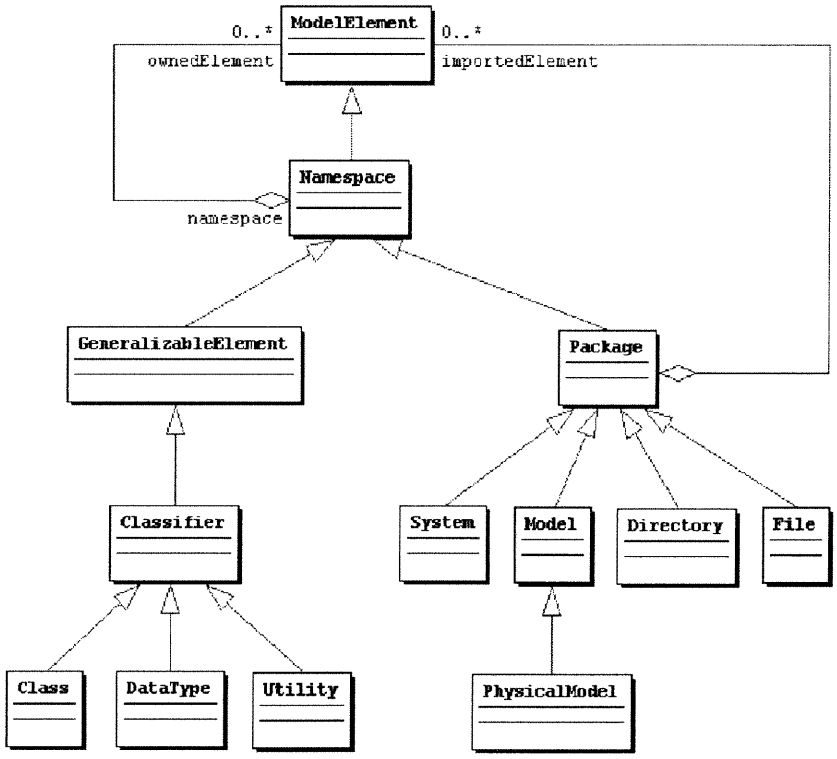


**Figure 2: SPOOL repository schema: Core classes**

The core classes of the SPOOL repository schema adhere to a large extent to the classes defined in the core and model management packages of the UML metamodel. As the elements of the system to be analyzed are stored under the format of the UML metamodel, they are called *ModelElements*, as defined in UML [38]. The core classes

define the basic structure and the containment hierarchy of the *ModelElements* managed in the repository (see Figure 2 and Figure 3). At the center of the core classes is the *Namespace* class, which owns a collection of *ModelElements*. A *GeneralizableElement* defines the nodes involved in a generalization relationship, such as inheritance. A *Classifier* provides *Feature*s, which may be structural *(Attribute*s) or behavioral *(Operations* and *Method*s) in nature (see Figure 3). A *Package* is a means of clustering *ModelElements*.

Moreover, only the classes defined at the bottom of the hierarchy are concrete and thus, only the Class, DataType, Utility, System, PhysicalModel, Directory and File classes of Figure 2 are instantiated, as are the Attribute, Operation, Method and Parameter classes of Figure 3. Each one of these classes represents a particular category of elements of object-oriented systems:

- *Class* represents a class, Directory represents a directory, and File represents a file.
- *DataType* represents the basic types of the language, for example int, float and char in C++.
- *Utility* is a concept to classify the elements that are not a part of a class, like global variables, or free functions.
- *System* represents a software system. System normally contains a physical model and a logical model (the logical model is not modeled in the schema of SPOOL).
- *PhysicalModel* represents the physical model of a system and contains the directories and the files of the system.
- *Attribute* represents the attributes of a class (or of a Utility if it acts as free variables).
- *Operation* is the signature of a method and contains a list of parameters (Parameter). In other words, a method has only one signature but a signature can be shared by several implementations and thus in various methods.

- *Method* is an implementation of an Operation and contains the body of the method. Several methods can implement the same signature (Operation).

- *Parameter* is a parameter preset in the signature of a method (Operation).

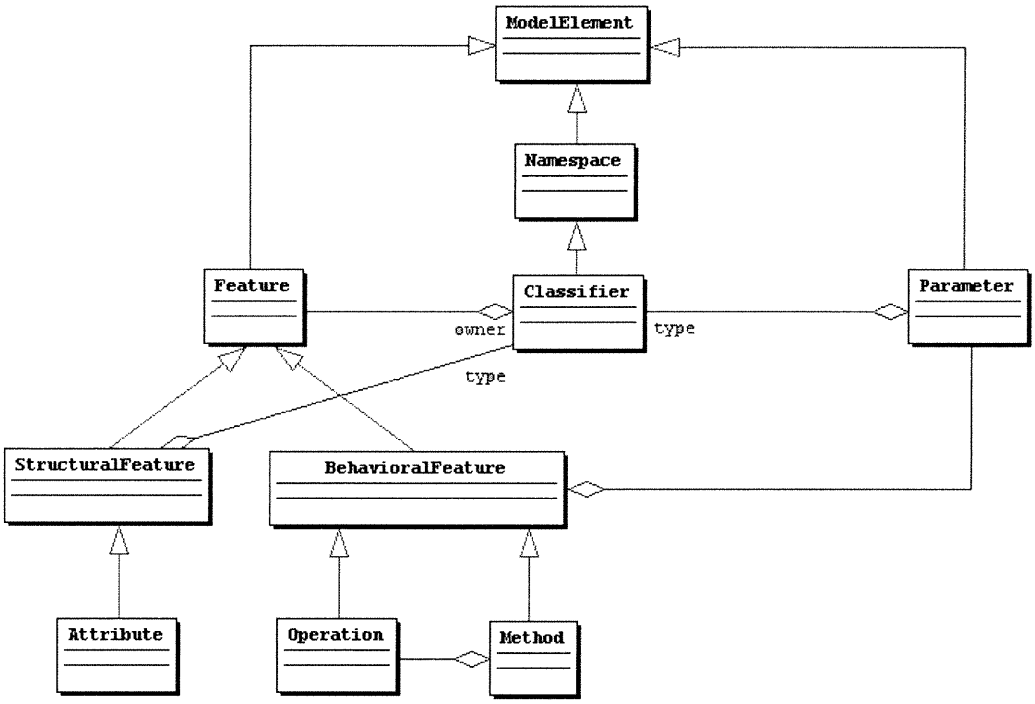


**Figure 3: SPOOL repository schema: Feature classes**

The preceding list classifies a subset of the elements of object-oriented systems and specifies their interrelationships. Figure 4 presents the SPOOL class hierarchy modeling the various types of actions. An Action is defined by UML as "An executable atomic computation that results in a change in the state of the system or the return of a value" [38]. In SPOOL, the actions are used to describe what is made in the body of the methods. For example, *CreateAction* models the instantiation of a class, and *CallAction* models an operation call. The other types of action (*ReturnAction*, *TerminateAction*, *DestroyAction* and *UninterpretedAction*) are not used at present by the SPOOL tools, and thus the information is not imported in the repository in order to reduce the size of the database and thus improve the performance of the analysis tools.

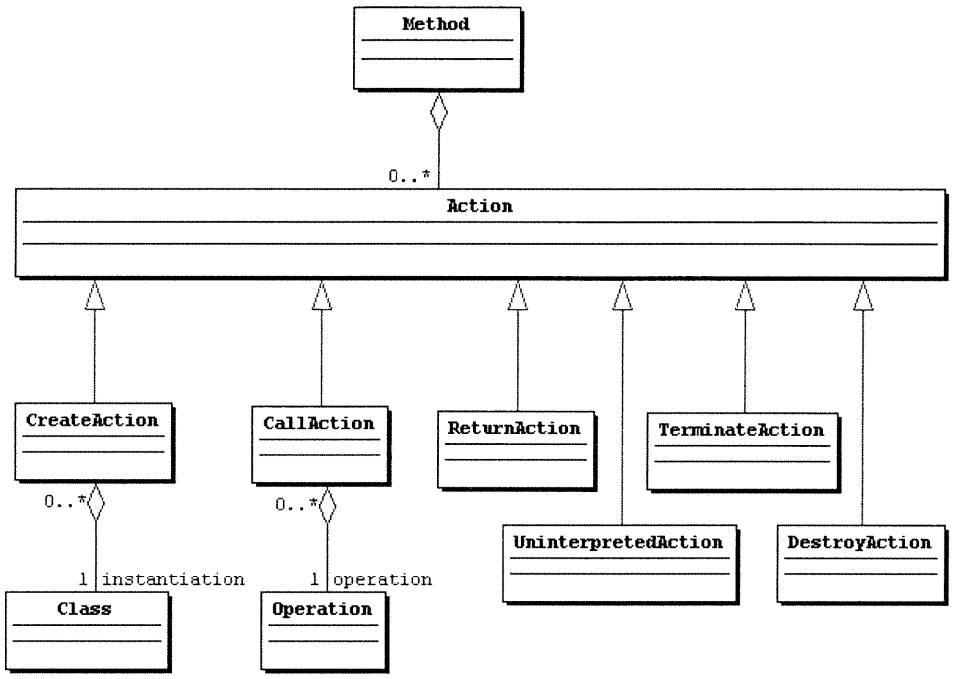


**Figure 4: SPOOL repository schema: Action classes**

The schema of the SPOOL repository described above is mainly based on the UML metamodel, but does not respect it 100%. Indeed, the UML metamodel is not conceived in terms of reverse engineering, but in terms of software design. Therefore, certain aspects of the underlying programming languages had to be added and certain modifications had to be made for considerations of performance.

## 3.3 The SPOOL Analysis Tools

The SPOOL environment provides a number of tools for investigation and visualization, allowing the exploration of a system at several abstraction levels (source code, structure, and design). Below, we will briefly present those analysis tools that were developed or simply integrated within the SPOOL environment.

### 3.3.1 Analysis at the Source Code Level

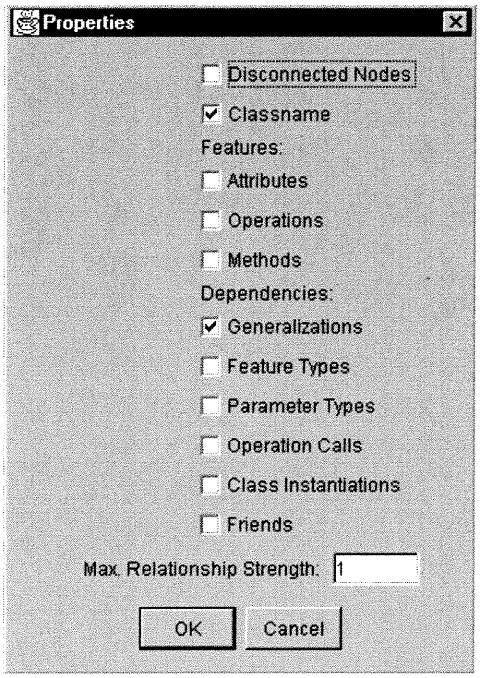


**Figure 5: Property sheet in UML diagrams**

The lowest level of abstraction at which the user can explore a system is the source code itself. Before the *Context Viewer* was integrated into the SPOOL environment, the environment did not allow investigation of the source code directly, since no editor had been implemented yet. However, there is a mechanism to integrate SNiFF+ [29] development environment into SPOOL. This mechanism makes it possible to visualize the elements of the source code via the various tools of SNiFF+ (editor of source code, hierarchy browser of classes, etc). Notably, this bridge between SPOOL and SNiFF+ allows accessing the source code, adding the power of SNiFF+ for software system inspection and navigation of the SPOOL environment.

### 3.3.2 Analyses at the Structure Level

There are normally two types of structures when we visualize software. The first is the physical structure of the software, such as the location of the source files in a hierarchy of directories, the location of the classes in the system, etc. The second structure concerns

the logical location of system elements; for example, the location of the various classes in an inheritance tree, or simply the location of a method in a function call diagram. The SPOOL environment supports UML diagrams to visualize six structural aspects of the software and to combine physical and logical structures. Via the property sheet, associated with UML diagrams (see Figure 5), all the association relationships stored in the repository, such as generalization, instantiation, aggregation, operation call, and friend, can be visualized in both separate and combined forms.

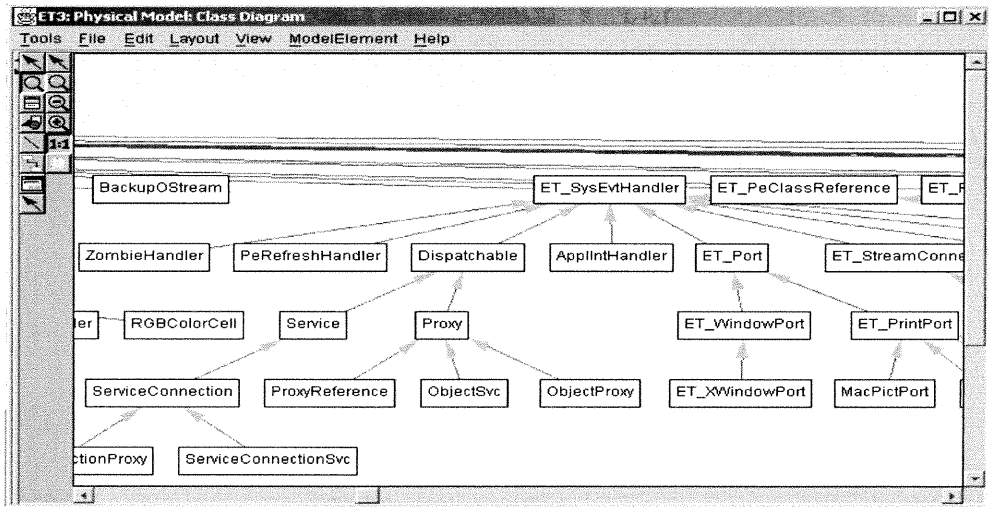


**Figure 6: Inheritance dependency diagram**

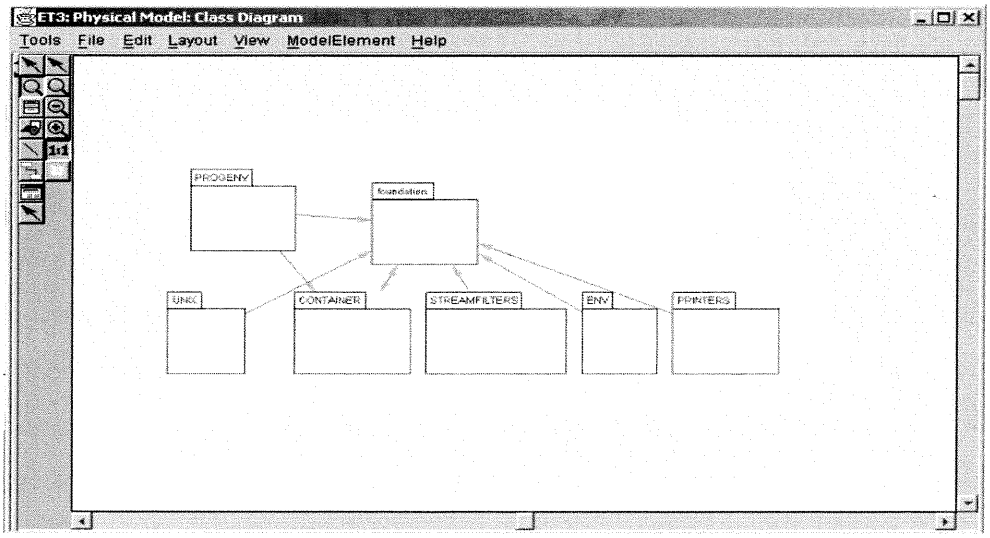


**Figure 7: Higher level dependency diagram**

By displaying the iheritance dependencies between classes, Figure 6 shows a part of the class diagram of the system ET++ [10], which is a system that comprises about 720 C++ classes distributed over more than 480 files. The environment also provides an aggregation mechanism to visualize higher-level dependencies. For example, class inheritance can be accumulated at the level of files or even directories, as shown in Figure 7. This approach allows one to extract certain information related to the design of a system, and to even detect certain problems; for example, an unforeseen coupling between two directories or two files.

### 3.3.3 Analyses at the Design Level

To fully comprehend the purpose of a given piece of software, the mere understanding of the static structure of the source code, or even the clear representation of the system's physical and logical structure, are still insufficient. The comprehension of the rationale behind design decisions is as important as the understanding of the software's structural and logical constituents. Design patterns capture the rationale behind recurringly proven design solutions and illuminate the trade-offs that are inherent in almost any solution to a non-trivial design problem. Design pattern recovery is supported in the SPOOL environment by two tools; namely, *design pattern detector* and *design inspector*. The former allows the user to detect the structure of design components (the pattern-like structures to be discovered), which are possible instances of design patterns; the latter makes it possible to inspect discovered design components [15]. It is inherently difficult to automatically recover high-level design components (for example architectural patterns [3]). SPOOL rather supports the recovery of design patterns that have a simpler structure, notably certain patterns of Gamma *et al.* [9]. These patterns are relatively low-level and detectable by static analysis; that is, no information about the dynamic behavior of the system is required (An example of a highly dynamic pattern is *Chain of Responsibility* [9], which can only be detected during runtime). The *design pattern detector* carries out queries on the SPOOL repository and detects certain structures corresponding to the implementation typically used for the selected pattern. Once the execution of a detection query is finished, the results must be visualized to allow a more

in-depth inspection by the user. In SPOOL, the visualization of the results of a query is done directly in the UML class diagram of the system. In SPOOL, each of the supported abstract design components comprises a so-called reference class. This is the class in the component's structure diagram that is considered most characteristic of the component's nature. Upon design recovery, we draw incremental bounding boxes around the reference classes of the implementations of an abstract design component. For example, Figure 8 shows the *Factory Methods* [9] found in the system ET++. This way of visualizing and organizing the information is very useful because it gives a first impression about the number and location of the possible instances of the specific design pattern in the system.

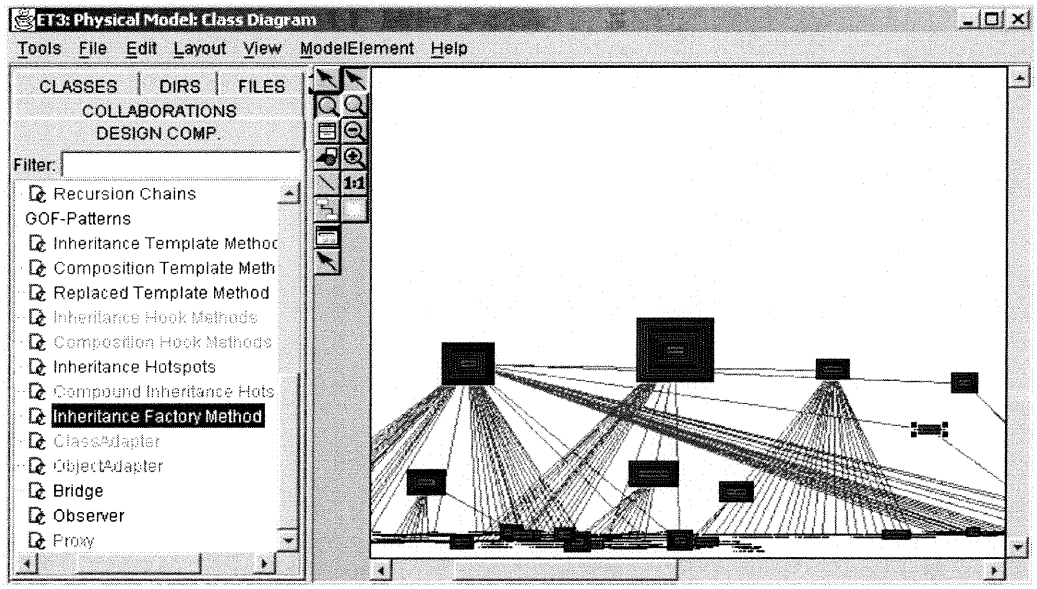


**Figure 8: Design pattern detection**

Supported by the *design inspector,* as shown in Figure 9, it is then possible for each of the detected design components to be inspected in a separate diagram, which displays information in three sections. The top section displays a list where all the instances of the pattern in which the selected element is the reference class are shown. The center section displays, in the form of a collaboration diagram, all the elements taking part in the pattern instance selected in the first section. Finally, the section at the bottom highlights each element of the center section in the context of the class hierarchy, which makes it possible to identify the components' relative locations.

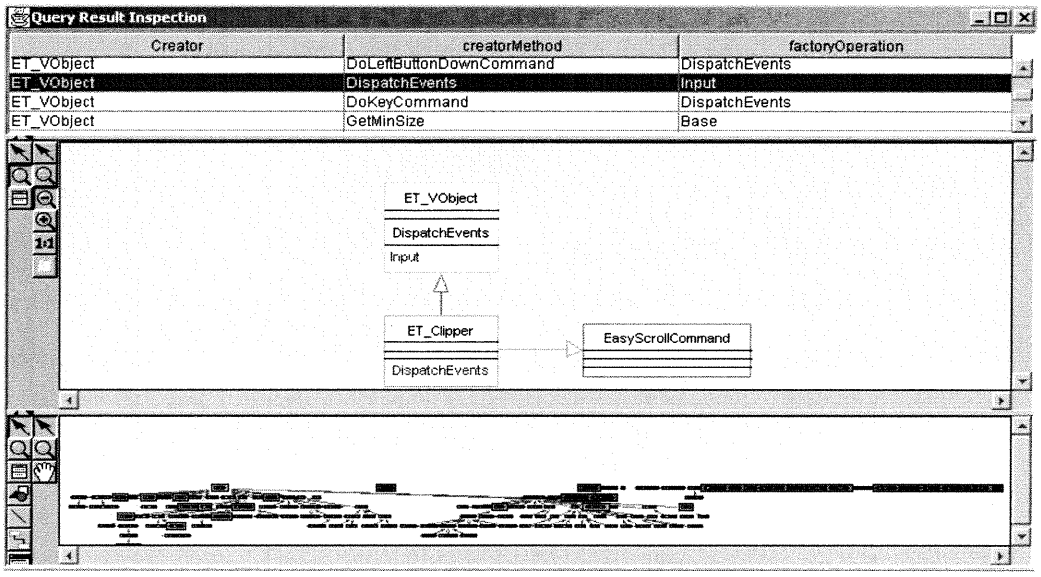


**Figure 9: Design inspector**

With the help of the various tools of the SPOOL environment, a user can detect certain structures in the system and inspect them to determine if they really form a part of a design component or not. In fact, an instance detected by the SPOOL environment is not necessarily a real design pattern instance, that is, one that exhibits the intention of that pattern. But the user can often get significant information, especially if the structure is relatively complex like *Bridge* or *Observer* [9] for instance.

## 3.4 Design Navigation

Design navigation in the SPOOL environment is supported by two tools: the *Design Browser*[19] and the *Retriever* [19]. The former allows for browsing the source code model and exchanging results with the different SPOOL tools; the latter supports full-text and structural searching.

The user interface of the Design Browser is separated into three sections: *Starting point*, *Queries,* and *Results* (see Figure 10). The first and the last sections each contain a list of *ModelElements*, characterized by their names and by differently colored icons that represent their kind (such as *C* for class, *F* for file etc). The list in the center of the

*Design Browser* contains the list of the queries supported by the tool according to the selected *ModelElements*. After the query is executed, the result is shown as a new list of *ModelElements* in the *Results* section. For example, Figure 10 shows the browser after the execution of a query that retrieves the namespace of *SetModified*, a method from the system *ET++* [10]. It is simple to add new user-defined queries into the *Design Browser.*

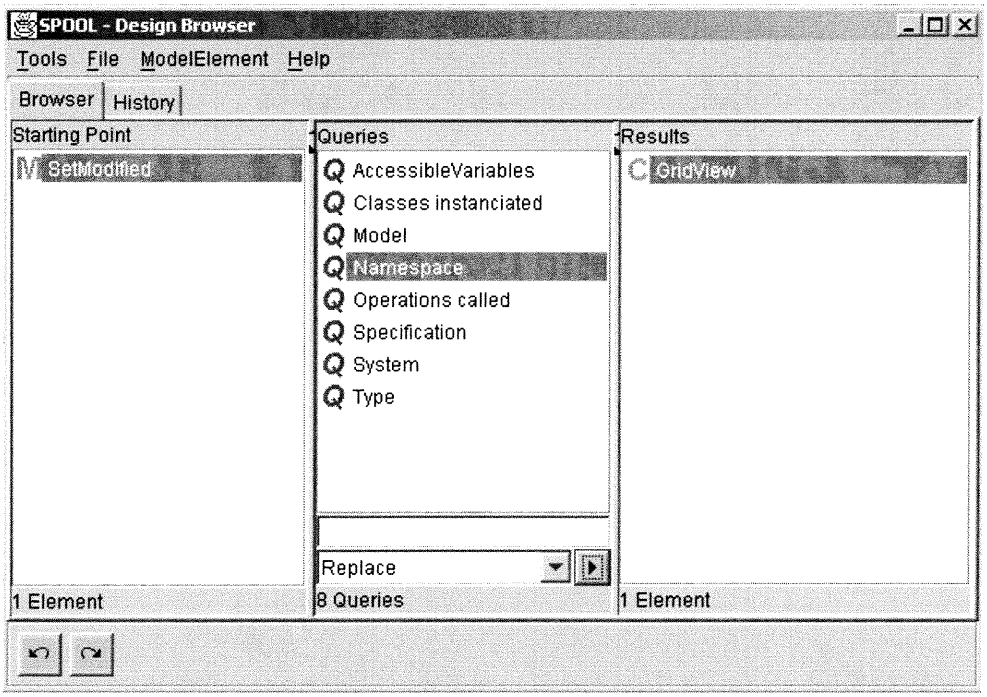


**Figure 10: Design Browser**

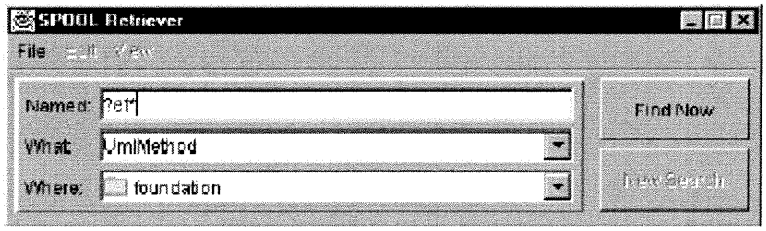


**Figure 11: Retriever**

The *Design Browser* is complemented by the *Retriever,* as shown in Figure 11, which allows the user to search for a string of characters in the names of the *ModelElements* that are contained in a namespace. Accordingly, a namespace class can be found in the

SPOOL schema, allowing the Retriever to search elements that are contained inside other elements, such as the *System*, *Packages*, *Directories*, *Files*, and *Classes*.

In the next chapter, we will present a new approach of software comprehension, namely *Visualization in Contexts.* The SPOOL environment, as presented above, lacks several important features to support this approach, such as a set of predefined context views, context customization, etc. For this reason, we integrated a prototype tool, namely the *Context Viewer,* into the SPOOL environment to support this approach.

# Chapter 4 : Overview of the *Context Viewer*

In Chapter 2, several cognitive models of comprehension were presented. We will present an approach of comprehension in Section 4.1; namely, *Visualization in Contexts*, which is based on certain aspects of these models as well as on concepts drawn from the object-oriented paradigm and from design patterns. The approach is strongly coupled with the utilization of the SPOOL environment presented in Chapter 3. It also demands new functionalities that need to be supported, which are described in Section 4.2. A new prototype tool, the *Context Viewer,* is introduced into the SPOOL environment to complement the support for our approach. We will go through the principles of its feature design in Section 4.3 and its main functionality in Section 4.4.

## 4.1 Our Approach: *Visualization in Contexts*

In order to facilitate the comprehension of the software, the user needs high-level information about the system under investigation, such as information at the structural and the design level. The automatic or semi-automatic recovery of this information may lead to a faster and more flexible comprehension approach than the various approaches of comprehension presented in Chapter 2.

This new approach is what we call *Visualization in Contexts,* which is an approach that supports several software comprehension directions, such as top-down, bottom-up or a mixture of both. Moreover, it allows for iterative comprehension based on the resolution of hypotheses. The tools that implement this approach must facilitate rapid navigation between various levels of abstraction, so that it is possible to better support the approach described by Von Mayrhauser and Vans [43][44].

*Visualization in Contexts* requires a set of predefined context views. These context views can be of structural nature such as inheritance relations between several classes, or can be directly related to the design of the system, like the elements' roles within one collaboration of a design pattern. During software comprehension, ambiguities are often caused by inadequacy or lack of information in the programmer's mental model of the software system. Such deficiencies can often be resolved by moving between the various abstraction levels of the system and by proving the hypotheses generated in one context view in another context view.

## 4.2 Requirements for the *Context Viewer*

The various comprehension tools presented in Chapter 2 do not provide the functionality required to support the *Visualization in Contexts* approach. In fact, some of the tools make it possible to visualize certain information in a predefined context (for example, in SNiFF+ one can visualize a class in its inheritance hierarchy), yet they are rather limited in that only a few contexts are supported and little design level information is provided. In addition, it is impossible to create personalized contexts from these tools.

The SPOOL environment already supports *Visualization in Contexts* to a certain extent. It handles the elements of a software system in an abstract way and supports the search and navigation for these elements in an efficient way. Moreover, it supports the automatic or semi-automatic recovery of various aspects of the system at the structural level and the design level. Still, SPOOL lacks certain functions demanded by the *Visualization in Contexts* approach, such as the following:

1. To visualize a small set of system elements in various precisely defined contexts.

2. To allow the creation of personalized contexts where only the desired elements are visualized.

3. To rapidly move between various contexts at different abstraction levels.

To complement the support to this comprehension approach, the new tool, *Context Viewer,* must provide a set of views for one *ModelElement* alone or a small number of *ModelElements* together in precise predefined contexts, which must focus on various abstraction levels. It must allow the user to customize the set of *ModelElements* under investigation according to his or her needs, and the various context views are static in respect to these *ModelElements*. Moreover, every piece of information shown in these views should carry a reasonably high amount of data about the role of these *ModelElements* without adding too much complexity to the view. Finally, the user needs a mechanism to more rapidly move between various contexts in order to facilitate mental models cross-referencing.

## 4.3 Principles of Feature Design

According to the requirements set out for the *Context Viewer,* we worked out four principles for its feature design. Each view in the *Context Viewer* was designed according to the following four design principles:

1. A context view of *Context Viewer* is meant to give a view of one *ModelElement* alone or a small number of *ModelElements* together, in precise predefined contexts.

2. Every piece of information shown in these views should have a direct relation to the subject *ModelElements* of the current predefined context.

3. Every piece of information should further describe the roles of the subject *ModelElements* of the context without adding too much complexity to the view. Providing the right amount of relevant information is key to the effectiveness of the *Context Viewer.*

4. The views proposed by the *Context Viewer* are static with respect to the set of *ModelElements* under investigation. In other words, the information that the analyst sees in a context view is determined solely by the *ModelElements* currently being

examined, and nothing apart from changing these *ModelElements* can change what is displayed in the view.

## 4.4 Main Functions of the *Context Viewer*

In Section 4.2, we have discussed the fundamental requirements of the *Context Viewer*. With these requirements in mind, we will briefly describe the main functions of the *Context Viewer*, which are implemented to satisfy those needs. The *Context Viewer Specification* in the Appendix presents further details about the tool's functionality. In this section, we will first introduce the so-called *e-set element*s and their supported operations. Then, we will go through each context view supported by *the Context Viewer.* The context views are organized into three categories; namely, source code view, structure views, and design views, each of which represents one abstraction level respectively. Finally, we will describe two supplementary mechanisms embedded inside the *Context Viewer*, namely *History* and *Synchronization.*

### 4.4.1 *E-Set* Elements and Operations

Before we go any further, we need to introduce the term *e-set element*s. In the *Context Viewer*, *e-set element*s refer to a set of *ModelElements* that are under examination. *E-set elements* can be manipulated by *add, replace,* and *remove* operations, and there are various ways to invoke these operations; for example, via popup menus, pull down menus and hot keys. This allows for the user to easily change the content of the current *e-set* to personalize visualization in the various context views.

### 4.4.2 Various Context Views

In the graphic user interface (GUI ) of the *Context Viewer*, as shown in Figure 12, various context views are organized as a set of nested panels, each of them containing a context view or a group of related views, which are available at the various levels of abstraction supported by the SPOOL environment. The kind of contexts supported is generally closely related to the underlying programming paradigm (object-oriented in our case) and

to the language used (C++ in our case). Each of these views is meant to visualize a small group of *ModelElements* in a precisely predefined context view. Figure 12, for instance, shows the *Source Code* context view.

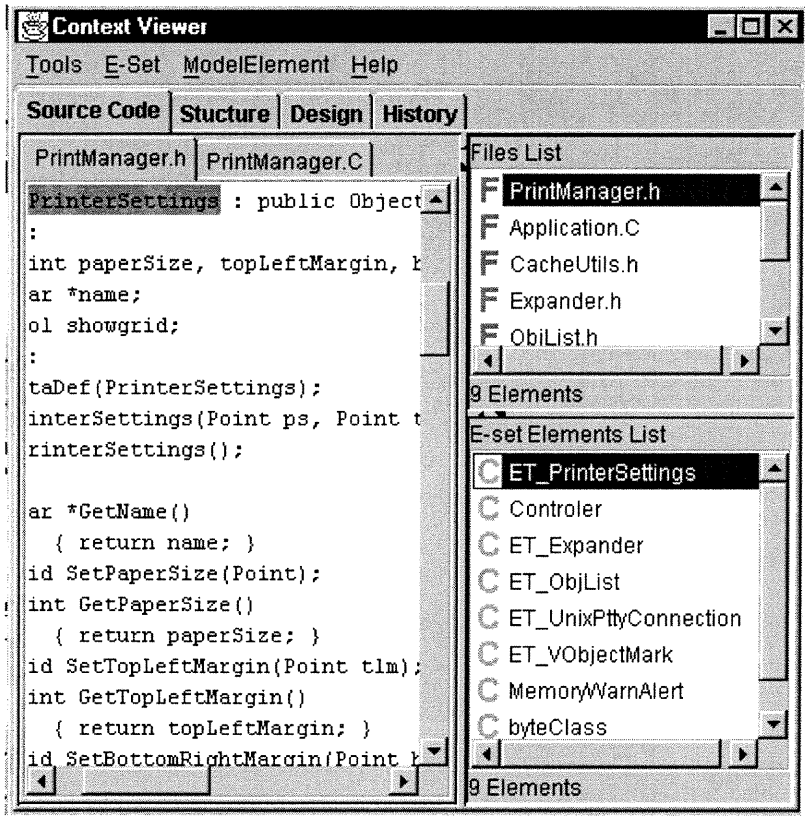


**Figure 12: Source Code context view**

### 4.4.2.1 Source Code Level Context

The most basic context view is obviously the source code itself. The *source code* view visualizes the piece of code relevant to the selected *e-set element*. This context view presents information about *e-set element*s at the lowest level of abstraction. Since SPOOL is targeting software written in C++, both the declaration and the definition of a *ModelElement* are queried and presented in this context view. As shown in Figure 12, for example, the declaration of class *ET_PrinterSettings* is highlighted in the source code of file *PrinterManager.h*, and the user can find its definition in file *PrinterManager.C*. It

facilitates comprehension by automatically locating and visualizing specific code segments in large-scale software.

#### 4.4.2.2 Structure Level Contexts

The *Context Viewer* supports six context views in this category, namely *Containment, Inheritance, Polymorphism, Call Action, Create Action,* and *Friendship*. Among these context views, only *Containment* describes the physical structure of the *e-set element*s; all others are related to their logical structure: *Inheritance, Polymorphism, Call Action, Create Action* are related to the OO paradigm; *Friendship* is specifically related to the C++ language.

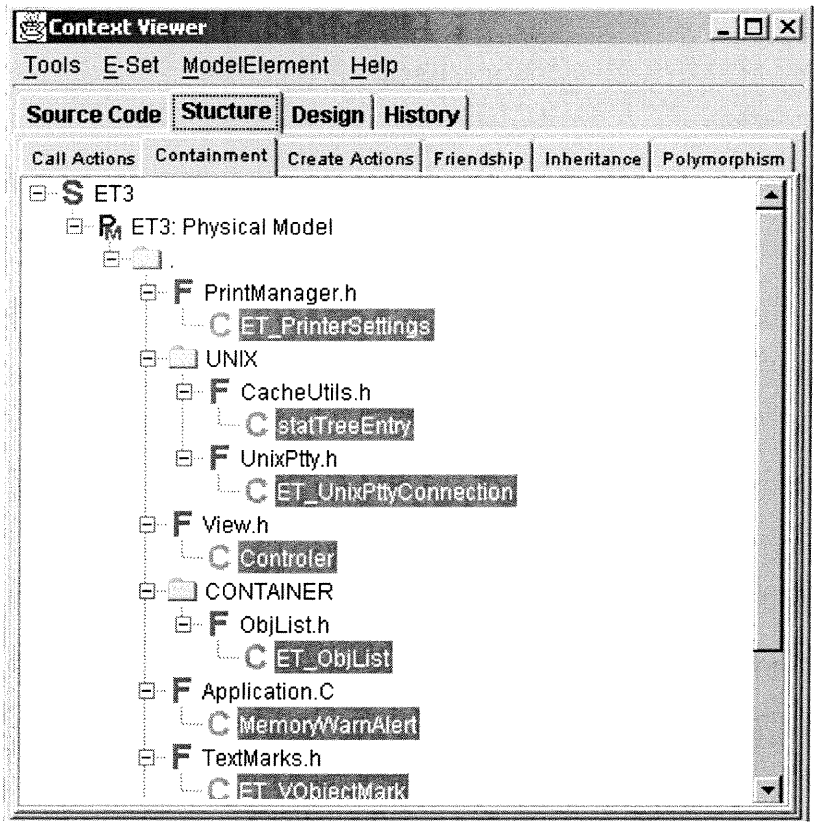


**Figure 13: Containment context view**

- *Containment* makes it possible to visualize the physical location of the declaration of an *e-set element*. The example presented in Figure 13 shows, in the form of a tree, the

location (in terms of which file, which directory, etc.) of certain classes of the system. In other words, *Containment* shows the hierarchical position of an *e-set element* and shows its location in the physical structure of the software at hand.

- *Inheritance* is a context view where the concept of inheritance is extended. Usually, inheritance is a relationship between classes; in the *Inheritance* context view, it can be examined at three structure levels; namely, the class level, the file level, and the directory level. Moreover, it is capable of visualizing the inheritance relationship between levels. As shown in Figure 14, for instance, it visualizes the inheritance dependencies at the directory level for *foundation* and *CONTAINER* and at the class level for *ET_Object, ET_Command,* and *MultiCellSelector*. It is also shown that the classes *ET_Object* and *ET_Command* have inheritance dependencies with these two directories respectively.

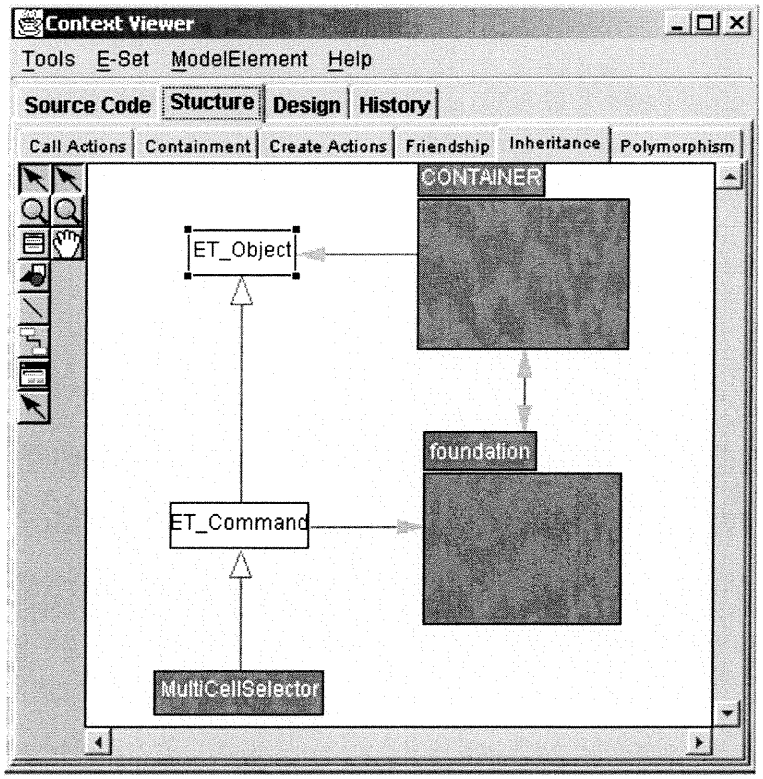


**Figure 14: Inheritance context view**

- *Polymorphism* visualizes the polymorphism information about the *e-set elements*. Given a method/operation that takes part in a relationship of method/operation overriding, *Polymorphism* is capable of visualizing where and how the method overriding happens in a UML style class diagram.

- *Call Action* is a context view where the concept of function call is extended. Usually, a function call is the relationship between a method/operation pair; in the *Call Action* context view, it can be examined at four structure levels, namely, the method/operation level, the class level, the file level, and the directory level. In addition, it is capable of visualizing the function call dependency between levels.

- *Create Action* is a context view where the concept of object instantiation is extended. Usually, object instantiation is the relationship between classes; in the *Create Action* context view, it can be examined at four structure levels, namely, the method/operation level, the class level, the file level, and the directory level.

- *Friendship* is a context view closely related to the programming language C++. It makes it possible to visualize the accessibility of certain *ModelElements* by the others in the context of a friendship relation, which is defined in C++.

#### 4.4.2.3 Design Level Contexts

The SPOOL environment allows for the detection of various concepts related to the design; most notably, certain design patterns [9]. The user can carry out certain queries about the system, in order to find particular structures corresponding to the implementation typically used for the selected pattern in the system. Since the result of these queries can be stored in the repository of SPOOL, the *Context Viewer* is capable of checking whether a specific *ModelElement* participates in one or several of these structures and visualizing the result. The *Context Viewer* currently supports, at the design level, four context views corresponding to the design patterns *Proxy, Factory Method, Observer,* and *Template Method,* respectively, as well as the *Multiple* view. This latter view allows for the visualization of *e-set* elements that participate in instances of two or

more different design patterns. In this section, we will briefly introduce the *Factory Method* view, *Template Method* view, and *Multiple* view. For a more detailed description of the various context views, please refer to the Appendix.

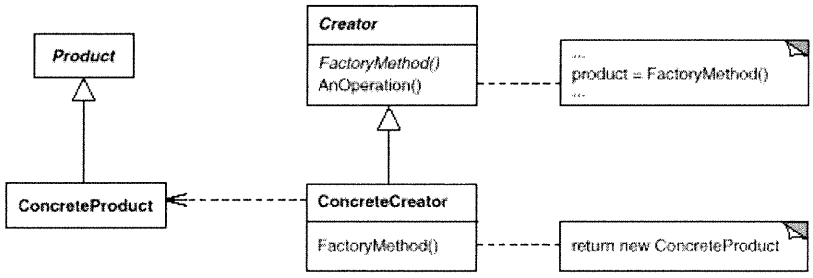


**Figure 15: Structure of Factory Method design pattern**

Figure 15 shows the UML class diagram of the structure of the *Factory Method* design pattern, which is one of the creational patterns as defined in the design pattern catalog [9]. This pattern provides a flexible solution for object instantiation. It allows a class to defer the operation of instantiation to be implemented in the subclasses.

We can see that there are four so-called *roles* in Figure 15; namely, *Product*, *Creator, ConcreteCreator,* and *ConcreteProduct.* Knowing which role each class is playing, the user can easily understand the collaboration between the classes that form a design pattern instance. Each of these design views can identify and visualize which role an *e-set element* (class) is playing in terms of its corresponding design pattern, therefore largely accelerating the software comprehension process.

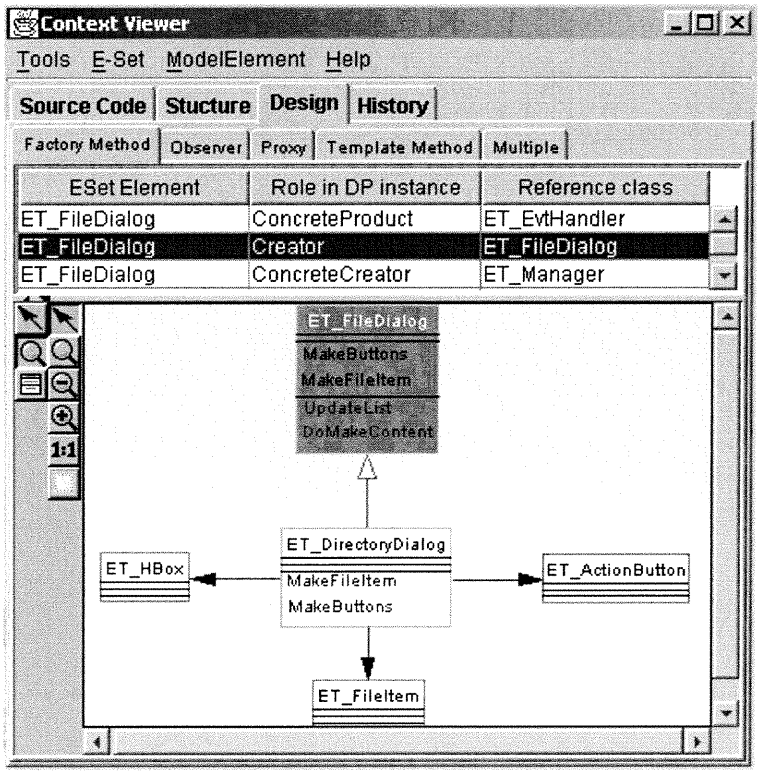


**Figure 16: Factory Method context view**

In Figure 16, the *Factory Method* view shows in its upper part the list of recovered instances of the *Factory Method* design pattern (DP), identified by the respective *E-set Element*, its *Role in DP instance*, and the so-called *Reference class*. The lower part shows the selected *Factory Method* pattern instance as a UML style class diagram. In the example presented in Figure 16, the user can easily tell that the *ET_FileDialog* plays the role of *Creator* in the selected pattern instance, the *ET_DirectoryDialog* plays the role of *ConcreteCreator* and the *ET_FileItem, ET_HBox, and ET_ActionButton* all play the role of *ConcreteProduct* in the *Factory Method* design pattern.

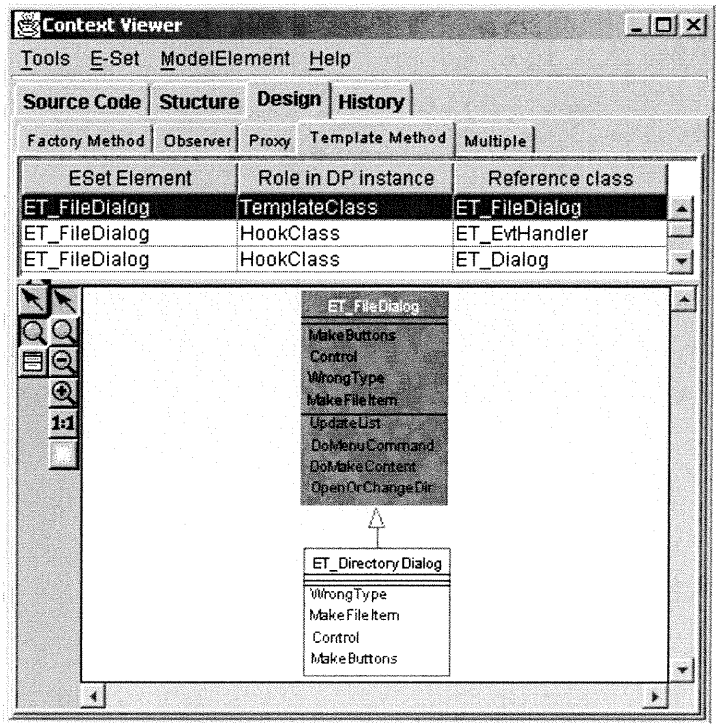


**Figure 17: Template Method context view**

The *Template Method* view visualizes the *e-set elements* in the context of the Template Method, which belongs to the catalog of behavioral design pattern. As an example, in Figure 17, the *Template Method* view shows that the *ET_FileDialog* participates in a instance of *Template Method* pattern and plays the role of *Template Class*. Within the *ET_FileDialog*, there are four template methods; namely, *UpdateList, DoMenuCommand, DoMakeContent, OpenOrChangeDir*. These methods invoke four hook methods: *MakeFileItem, MakeButtons, WrongType, and Control. ET_DirectoryDialog* plays the role of *Hook Class*. It inherits from *ET_FileDialog* and implements these hook methods.

*Multiple* is a context view that is able to visualize the *e-set elements* in the context of more than one design pattern. By the selection of multiple rows from the design pattern instances table, the user can investigate *e-set* elements in terms of any combination of the four design patterns *Factory Method*, *Template Method*, *Observer*, and *Proxy*. The example presented in Figure 18 shows that the currently selected rows form the

combination of the patterns *Factory Method* and *Template Method.* The current *e-set* element is *ET_FileDialog*. We know that it participates in both patterns from previous examples.

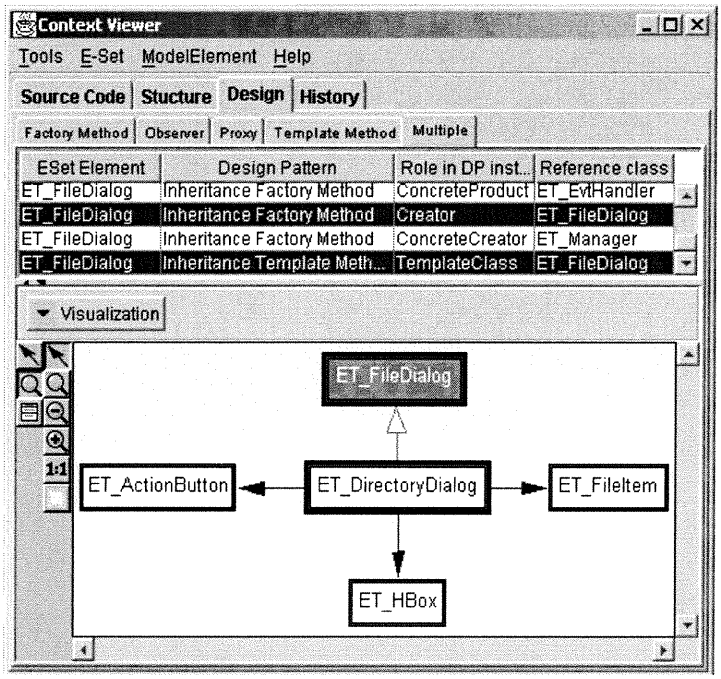


**Figure 18: Multiple context view**

In *Multiple* view, the participants for different patterns are shown in different predefined colors. If a class participates in more than one pattern instance, its bounding box will comprise more than one layer; each layer is presented in the color of its associated pattern, such as *ET_FileDialog* and *ET_DirectoryDialog*. The strength of *Multiple* is that the user can see how certain *ModelElements* collaborate with others that are participants of different design patterns in that view. To avoid confusion, there is no method or operation shown in the UML style class diagram; the *Multiple* simply tries to give the user an overall view of certain *ModelElements* in terms of design patterns. For more detailed information on each specific design pattern with which a specific *e-set element* gets involved, the user could go to the specific design pattern view.

### 4.4.3 Mechanisms Embedded inside the *Context Viewer*

There are two supplementing mechanisms embedded inside the *Context Viewer*, which largely facilitate and speed up the software understanding process, as we will explain below.

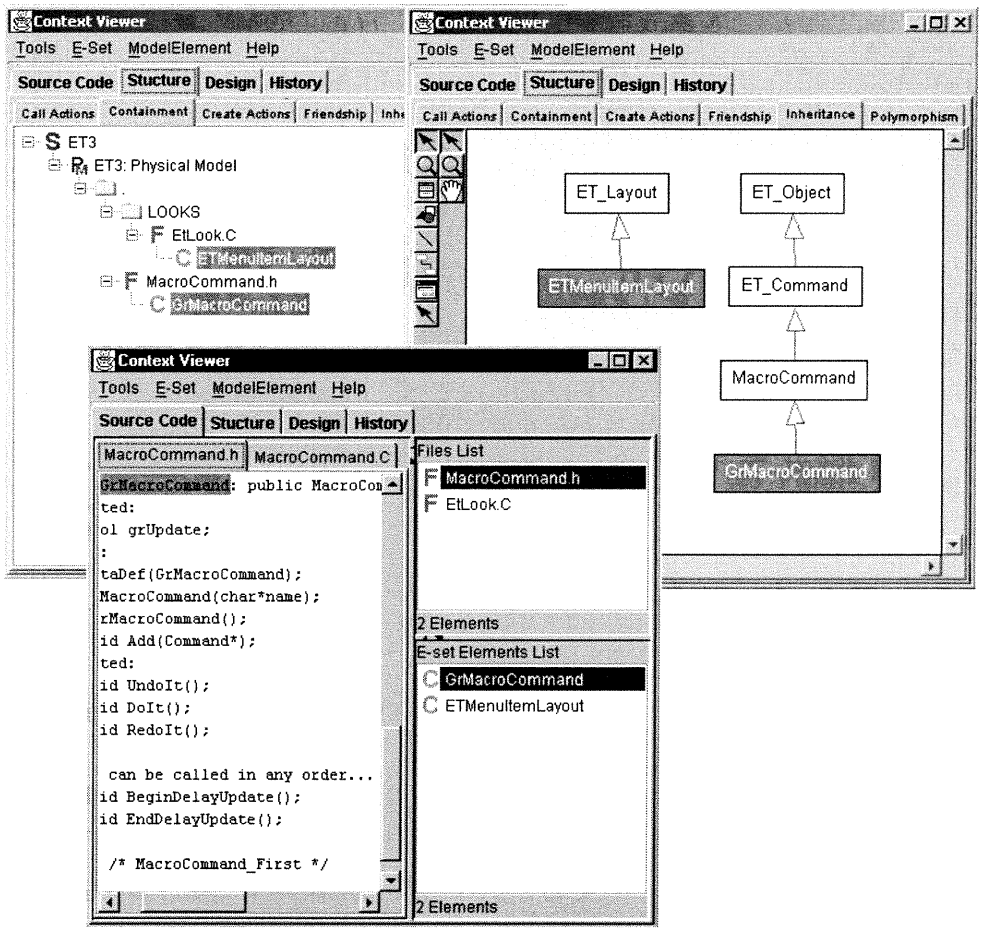


**Figure 19: Synchronization Mechanism**

#### 4.4.3.1 Synchronization Mechanism

The Synchronization mechanism makes sure that all currently available 12 context views are synchronized; that is, that they display the same *e-set elements* whenever the analyst selects a different context view. This mechanism is meant to reduce the analyst's cognitive overhead when cross-referencing mental models. During the investigation, to

switch between the various views at different levels of abstraction, the analyst simply has to click on the tabs that indicate the respective views.

Figure 19, for example, shows the *e-set elements ETMenuItemLayout* and *GrMacroCommand* in three context views; namely, *Source Code, Containment,* and *Inheritance.* To switch between these views, it only takes about three mouse clicks. Obviously, this reduces the user's cognitive overhead and greatly facilitates the cognitive process.

#### 4.4.3.2 History Mechanism

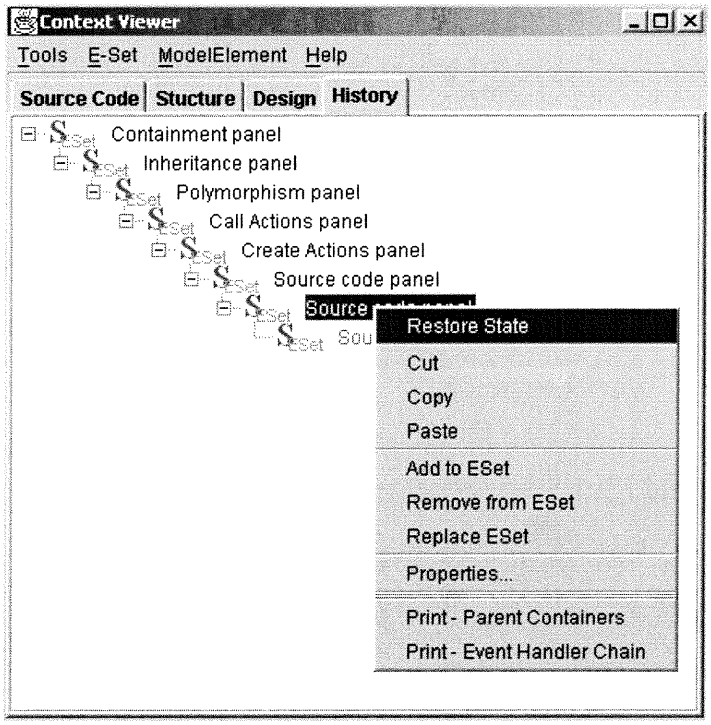


**Figure 20: History Mechanism**

Each instance of the *Context Viewer* stores all the states in which it has been since its creation as a history tree. A state in the *Context Viewer* is defined as the content of the *e-set* together with the last view the user has displayed with those *e-set elements*. The history tree can be accessed through the history tab. As shown in Figure 20, it is implemented as a vertical tree, each node representing a *Context Viewer* state, and the

current state is in magenta color. By selecting a state and invoking the command *Restore State,* the user can get back to previous states of the *Context Viewer* instance, so the user will never lose the experience acquired during one investigation.

# Chapter 5 : Design and Implementation of the *Context Viewer*

After the brief overview of the *Context Viewer* in Chapter 4, we will review in this Chapter the considerations that led to the design of the *Context Viewer*. There were many factors and considerations involved during implementation in order to achieve a good quality in terms of object-oriented design.

Firstly, we will describe some factors that affected our design decisions. Then, in Section 5.2, we will present four design components [15] that are relevant for the design of the *Context Viewer*. In Section 5.3, we will share our experience on how to select a suitable software package automatic layout given our specific project needs. Finally, we report on the validation and testing of the *Context Viewer*.

## 5.1 Factors affecting the Design

The *Context Viewer* has been implemented within the SPOOL environment using the SPOOL reverse engineering framework and the JKit/Go visualization framework. A framework is a class hierarchy plus a model of interaction among the objects instantiated from the framework. In addition, a framework reverses the traditional idea of component reuse. Instead of a programmer writing a main program that calls on re-usable procedures, a programmer instantiates objects from the framework's class hierarchy and then provides methods for the framework to call; this is so-called reverse programming. Therefore, one major design guideline of the *Context Viewer* tool is to try to tailor the SPOOL reverse engineering framework and Jkit/Go visualization framework according to our needs, instead of introducing new classes and mechanisms that define the interaction between their objects. Whenever possible, we tailor these frameworks by

providing highly specialized routines that are called by them. For example, to handle user related events in the *Context Viewer*, we decided to take advantage of the *Event Handler* design component, which is a mechanism about how to handle requests. Its details are presented in Section 5.2.4.

Another main design guideline for the *Context Viewer* is to adopt design pattern solutions [9] for the design problems at hand in order to design a reusable object-oriented application with good quality. Moreover, one of the major strengths of the SPOOL environment is design pattern recovery; the *Context Viewer* has a set of design views that are design pattern related, so it is in our interest to make our tool rich in design patterns by nature.

## 5.2 Context Viewer Design Components

There are many design components in the *Context Viewer*. In this section, we present four major design components in order to demonstrate the design quality of the *Context Viewer*. They adopt solutions of four design patterns [9]: the *ESet observer* for *Observer*, the *Text Search Strategy* for *Strategy*, the *View Factory* for *Abstract Factory*, and the *Event handler* for *Chain of Responsibility*. For each design component, we will describe the design problem on which it focuses, the solution for that design problem, its structure in the UML style class diagram, and its participants.

### 5.2.1 ESet Observer

*ESet Observer* is a design component whose objective is to establish a one-to-many dependency between an *ESet* object and different context views, so that all context views are notified and updated automatically once the *ESet* object changes its state.

**Problem**:

For each *Context Viewer* instance, there is only one *ESet* object. The visualization content in all the views in this instance will depend on the content of the *ESet* object in order to achieve synchronization among those views. In other words, once the state of the *ESet* object changes, all the views should update themselves accordingly. This also means that any change made to the *ESet* object in one view will in turn affect the other views' visualization. Moreover, it should be flexible so that adding new views and removing views from the *Context Viewer* will not require a lot of effort.

**Solution**:

The intention of the design pattern *Observer* [9] is to define a one-to-many dependency between objects so that when one object changes its state, all its dependents are notified and updated automatically. For the design requirement of the *Context Viewer*, we adopt the solution described by the *Observer*. As shown in Figure 21, it specifies that the subject class is observed by the observer classes, which can be attached to or be detached from the subject class. When the *ESet* object changes its state, it notifies all the observer classes by calling their method react().

**Structure:**

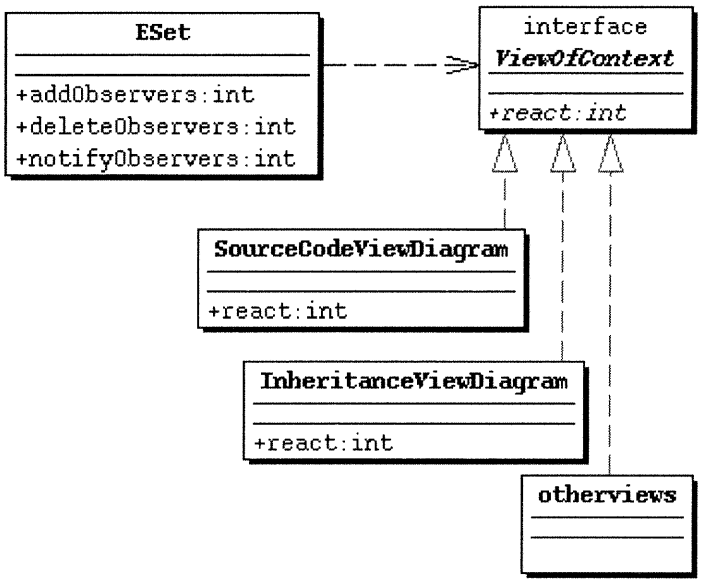


**Figure 21 : ESet Observer**

**Participants**:

*ESet* is the subject class observed by the *ViewOfContext,* which is an interface implemented by all the views contained in the *Context Viewer.* All the views have to implement the method react().

### 5.2.2 Text Search Strategy

*Text Search Strategy* is a design component whose objective is to define a set of string search procedures, encapsulate each one of them, and let them vary independently from the client that uses them.

**Problem:**

For the Source Code View, the view at source code level, we need to find the declarations of *ModelElements* (for example, the declaration of a class or a method) so we can

highlight them out from the text pane in this view. Since there are many possible search algorithms that can do it, and we may want in the future replace the current algorithm by another one, we want to achieve the flexibility that different text search algorithms are interchangeable.

**Solution:**

The intention of the design pattern *Strategy* [9] is to define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. This is suitable for our need; to comply with the interface *TextSearchStrategy*, each concrete text search strategy implements the method *locate()* respectively, and therefore the client is not aware of the change if one of the concrete text search strategies is replaced by another one.

**Structure:**

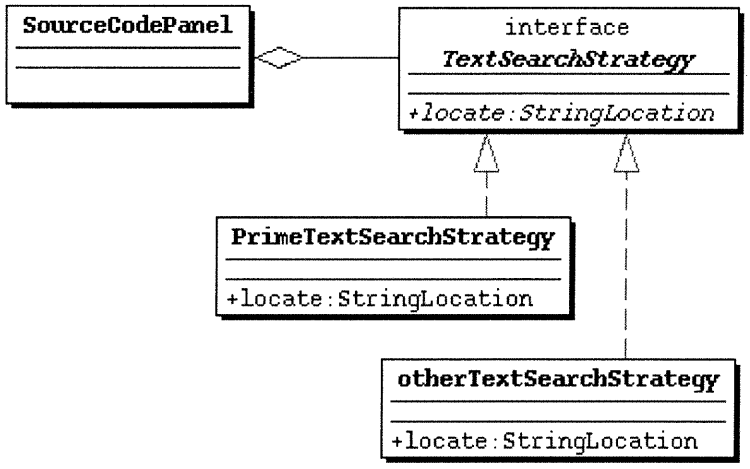


**Figure 22: Text Search Strategy**

**Participants:**

As shown in Figure 22, *SourceCodePanel* is the container of the *TextSearchStrategy*, which declares the common interface for all supported search strategies. The

*PrimerTextSearchStrategy* is one of those concrete strategy classes that implement a specific text search algorithm. The *SourceCodePanel* calls the algorithm implemented in method *locate ()* within the *PrimerTextSearchStrategy* or other text search strategies.

### 5.2.3 *Context Viewer* Factory

The *Context Viewer Factory* is a design component whose objective is for *the Context Viewer* diagram to create a set of context views without knowing the concrete classes of each view.

**Problem:**

In the *Context Viewer*, various design views are designed so that they are all composed into the *CVDiagram* during its initialization, which is the main container of these views. We need to enforce the constraint that these views are meant to be used together. At the same time, the *CVDiagram* should be independent of how its contained views are created.

**Solution:**

The intention of *Abstract Factory* [9] is to provide an interface for creating families of related or dependent objects without specifying their concrete classes. For the design requirement of the *Context Viewer*, we adopt the solution described by *Abstract Factory*. It specifies that the client only knows about the abstract factory and abstract product interface, each concrete factory class creates a set of concrete products.

**Structure:**

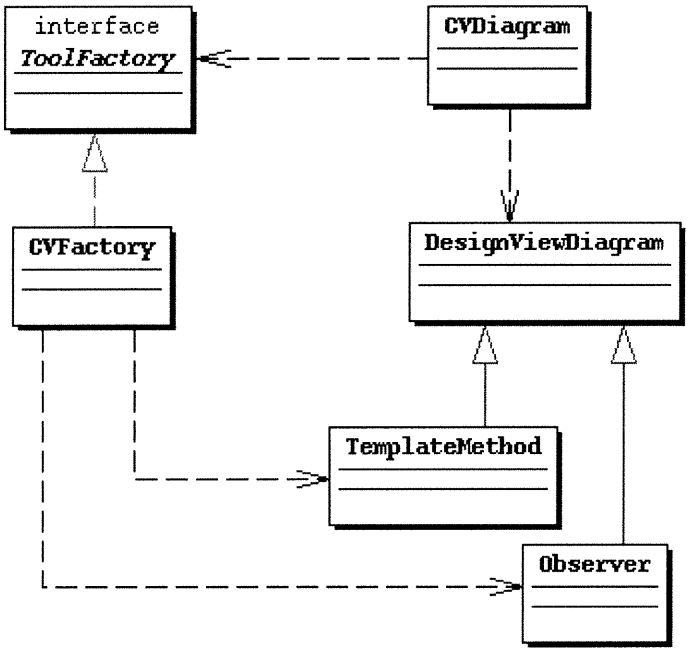


**Figure 23: Context Viewer Factory**

**Participants:**

As shown in Figure 23, the *CVDiagram* is only aware of the *ToolFactory* and *DesignViewDiagram. CVFactory* is the concrete factory that creates a set of design views; for example, *Observer* or *TemplateMethod,* etc.

### 5.2.4 Event Handler

The *Event handler* is a design component whose objective is to avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. In fact, *Event handler* is an embedded mechanism inside the SPOOL environment; *the Context Viewer* extends and takes advantage of this mechanism to handle *Context Viewer* specific events, such as *ESetEvent* and *CVHistoryDiagramEvent*.

**Problem:**

During investigation based on the *Context Viewer,* one thing the user can do is add/remove/replace the currently contained *ModelElements* in *ESet* object. Once such an event occurs, we want to handle it in a flexible manner: giving more than one object a chance to handle this *ESetEvent* and avoid hard coding the receiver of this event.

**Solution:**

The intention of *Chain of responsibility* [9] is to avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request; chain the receiving objects and pass the request along the chain until an object handles it. The SPOOL environment has an embedded mechanism, the *Event handler,* which is a design component and which adopts the solution of the *Chain of responsibility* design pattern. We can take advantage of the *Event handler* to implement our design for the needs of *the Context Viewer.*

**Structure:**

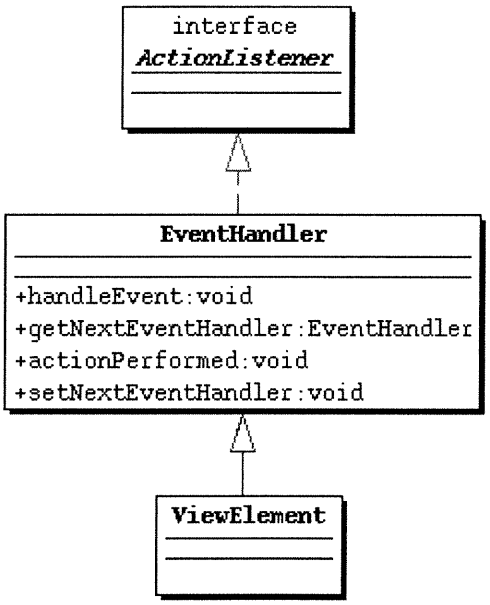


**Figure 24: Event Handler**

**Participants:**

As shown in Figure 24, the *EventHandler* plays the core role in this design component; it has the methods *getNextEventHandler*() and *setNextEventHandler*() to form the responsibility chain, and *handleEvent*() to react to the event. It also implements the *ActionListener* to benefit from the Java Event processing mechanism. The *ViewElement* subclasses from *EventHandler* so that all the *ViewElements* could be candidates to handle the events.

## 5.3 Choice of Layout Strategy Tools

During the implementation of the *Context Viewer*, we had to select a visualization tool to function as the layout strategy in order to generate the UML style class diagram visualization for design pattern views. Before the implementation of the *Context Viewer*, SPOOL employed *the Dot & Dotty* as the layout package to make a layout for all the UML style diagrams. Even though it works well, it has two main limitations: one is that *the Dot & Dotty* is not written in Java and the tools in the SPOOL environment have to use the *Runtime.exec()* method to create the native process to use it; this makes the SPOOL environment less portable. Another one is that it does not support the orthogonal layout strategy, which is arguably the most suitable layout strategy for the visualization of UML style class diagrams.

In order to overcome these limitations, we tried to find a better layout tool. Before we made our decision, we did a survey among the available visualization tools. The following is the result of the survey, focusing on the layout strategy and programming language support aspects:

**1. Visualization of Compiler Graphs (VCG)** [40]**:**

The *VCG* reads a textual specification of a graph and visualizes the graph. Its layout algorithm can be controlled in different ways.

- Supports orthogonal layout.
- *VCG* is written in C++.

**2. CodeCrawler** [4]:

*CodeCrawler* is a language and platform independent reverse engineering tool that combines software visualization and software metrics.

- No orthogonal layout support.
- *CodeCrawler* is written in Smalltalk.

**3. DaVinci** [5]:

*DaVinci* is a universal, generic visualization system for the automatic generation of high-quality drawings of directed graphs.

- No orthogonal layout support.
- *DaVinci* is written in C++.

**4. GRASP** [11]:

GRASP is a molecular visualization and analysis program. It is particularly useful for the display and manipulation of the surfaces of molecules and their electrostatic properties.

- No Orthogonal layout support.
- GRASP is written in FORTRAN.

**5. Visualizing Graphs with Java (VGJ)** [42]:

VGJ is a tool for graph drawing and graph layout. It supports three types of layout strategy: *Tree algorithm, Spring embedder*, and *Directed graphs.*

- No orthogonal layout support.
- VGJ is written in Java

**6. Tom Sawyer Graph Layout Toolkit (GLT)** [37]:

The Graph Layout Toolkit is a graphics system independent component that allows the visualization of relational data through the use of a sophisticated graph management system, object positioning libraries, and diagram editing APIs.

- Orthogonal layout support.
- GLT has a Java version.

Among the six visualization tools mentioned above, we selected the Java version of the Tom Sawyer Graph Layout Toolkit as the layout tool for the SPOOL environment, due to its support of orthogonal layout and its ability to enhance the portability of SPOOL.

## 5.4 Implementation and Experiences

The *Context Viewer* has been implemented in Java, using JFC/Swing components [13]. The implementation consists of some 70 classes and approximately 20,000 lines of code (LOC; comment lines not counted). All the functions of the *Context Viewer* as described in the Appendix have been implemented.

During the implementation of the *Context Viewer*, we have tested it with a medium-sized system, namely ET++ [10], which comprises about 720 C++ classes distributed over more than 480 files. Usually, if there were less than five *ModelElements* in an *e-set*, it took less than 30 seconds for *Context Viewer* to generate all the 12 context views (12 context views are generated at the same time to support the Synchronization mechanism, see Section 4.4.3.1). It may take considerably longer in case there are considerably more *ModelElements* in the *e-set*. Keep in mind, however, that the *Context Viewer* is designed to investigate a small amount of *ModelElements*.

Even though we have not yet tested the *Context Viewer* with very large software systems, we are confident with the performance of the *Context Viewer* for that kind of systems. The reason is that the SPOOL environment is the infrastructure for the *Context Viewer*. The SPOOL repository is the key part in the SPOOL environment concerning scalability, and it has already been shown that it can accommodate very large systems while maintaining performance at an acceptable level [20][24].

# Chapter 6 : Examples

In this chapter, we present three examples that demonstrate how the *Context Viewer* together with other SPOOL investigation tools can be used to support program comprehension. The system used for the examples is the application framework ET++ [10], a system that comprises about 720 C++ classes distributed over more than 480 files. The first two examples address context investigation and design pattern reengineering. The third example illustrates how the synchronization and history mechanisms reduce the analyst's cognitive overhead during the investigation.

## Example 1 : Investigation of Contexts

The investigation and visualization of contexts is key to program comprehension. The *Context Viewer* is designed for the quick retrieval and visualization of the various abstractions concerning the *e-set*, the group of selected *ModelElements.*

Figure 25 illustrates a typical investigation and visualization scenario. In this example, the analyst has found that *ET_FileDialog* is one of the classes of the system *ET3,* using the SPOOL Design Browser (window 1) as a query engine. A further query in the Design Browser reveals that *MakeFileItem* is one of the methods of class *ET_FileDialog* (not shown in the figure). Interested in further investigating *MakeFileItem*, the analyst specifies it as a one-element *e-set* by dragging and dropping the corresponding label into a running instance of the *Context Viewer.* In the *Containment* view of that instance, the physical location of *MakeFileItem* in terms of systems, packages, files, and classes is visualized (window 2). Then, after cloning the *Context Viewer* instance and selecting the *Create Actions* view (window 3), the analyst can see that *MakeFileItem* instantiates the class *ET_FileItem.* Furthermore, from the *Polymorphism* view (window 4), one learns

that the method is only overridden once in the class *ET_DirectoryDialog*. Object-oriented systems are difficult to comprehend because of the distribution of functionality and polymorphism, yet the comprehension task becomes much easier, in case such views are readily available.

Note that only the most relevant context information is visualized in each view. In this way, the analyst does not become overwhelmed with too much information. By navigating through the views pertaining to the structure level, the analyst can quickly grasp the physical and logical context of the method at hand. The analyst may then decide to continue the investigation and go to the *Source Code* view (window 5), where the definition and declaration of *MakeFileItem* is shown.

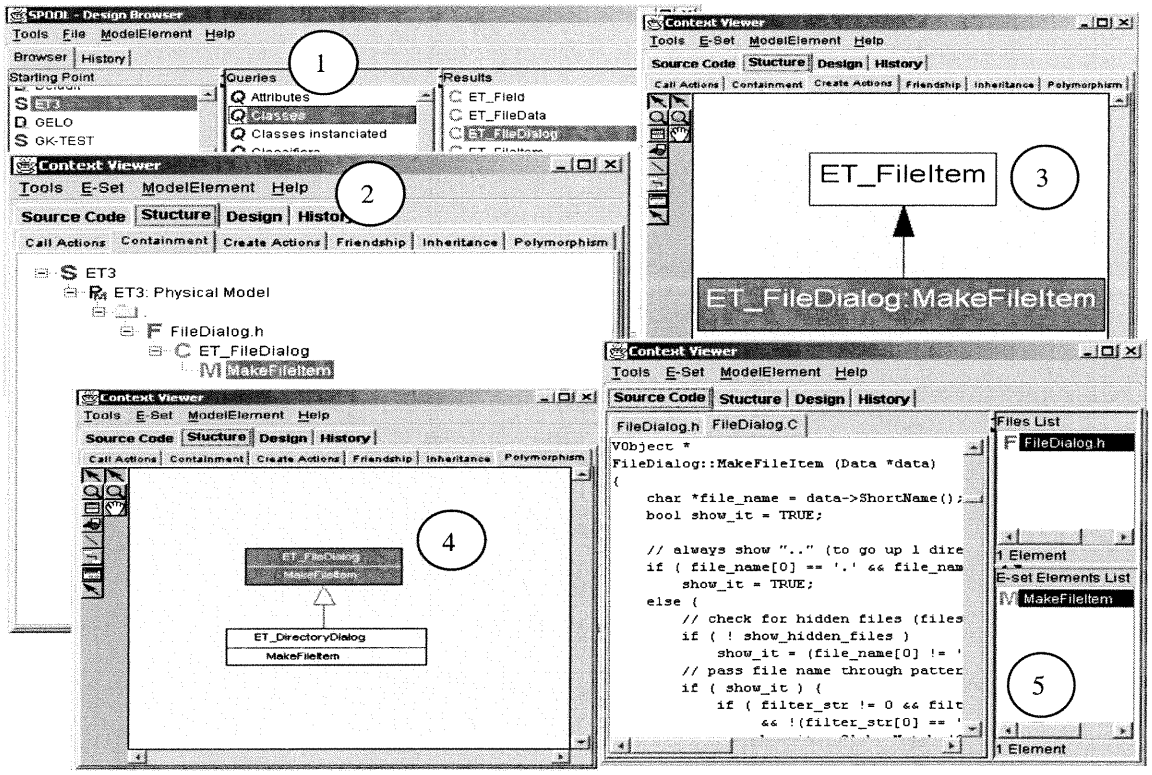


**Figure 25: Investigation of contexts**

## Example 2 : Reverse Engineering of Design Patterns

Quite often, the key design decisions in object-oriented systems are implemented based on well-known design patterns [3, 9, 22]. In [14, 21], we have discussed the use of SPOOL in reverse engineering some of these patterns. In this section, we illustrate how the recovery and analysis of design patterns can be further supported by means of the *Context Viewer*.

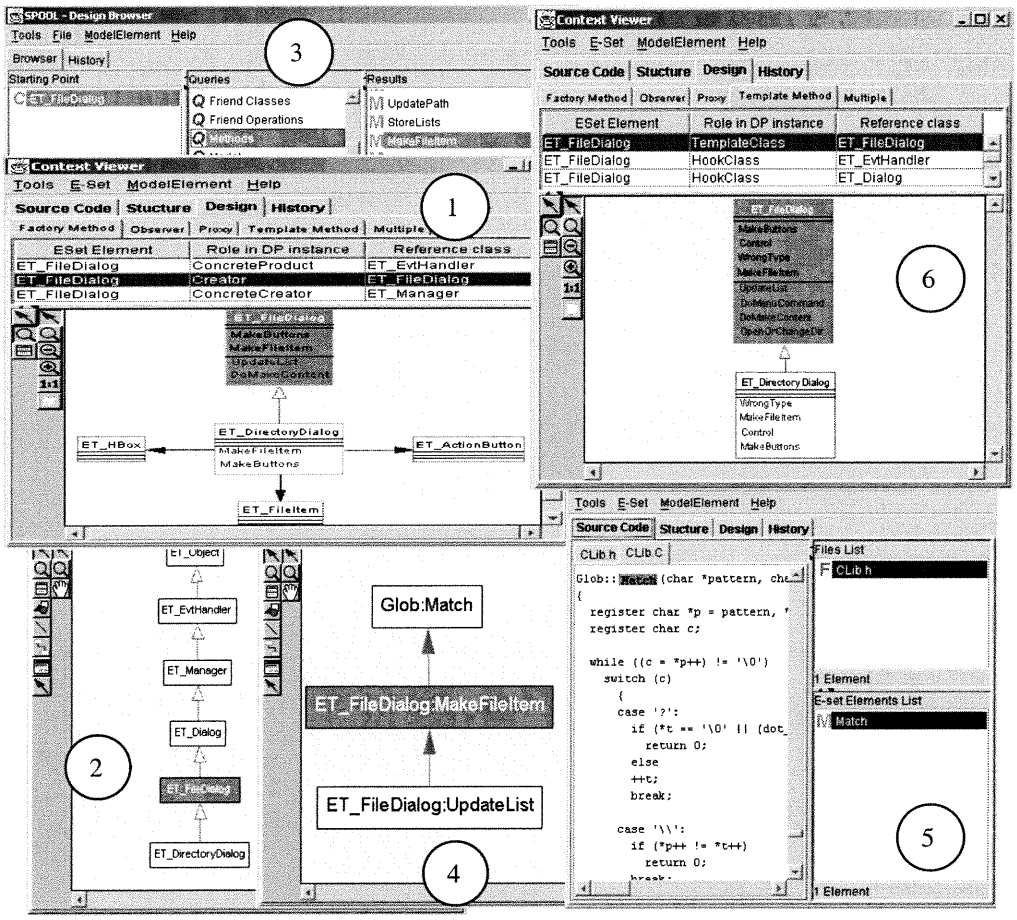


**Figure 26: Reverse engineering of design patterns**

As an example, window 1 of Figure 26 shows the *Factory Method* view related to *ET_FileDialog*, which happens to be the only *e-set* element. The upper part of the view indicates that *ET_FileDialog* participates in several instances of the *Factory Method* pattern. The lower part shows the selected *Factory Method* instance as a UML style class

diagram. The example shows that the class *ET_FileDialog* owns the method *UpdateList* and *DoMakeContent* that invoke the factory method *MakeFileItem and MakeButton*, which in turn are overridden in the subclass *ET_DirectoryDialog* of *ET_FileDialog* and instantiates three *concreteProducts,* namely *ET_FileItem, ET_ActionButton,* and *ET_HBox.*

The content of this view is automatically retrieved from information in the SPOOL repository that was generated by one of the SPOOL design recovery queries for the *Factory Method* pattern. Instead of showing all the information generated by a design query, the *Context Viewer* visualizes only the most relevant information related to the *e-set* elements. The design views provide precious information for program comprehension as they present in a concise way all the classes that take on a role in a pattern-based collaboration. Note that in the physical file structure, these classes may be spread out over many directories and subsystems.

To further investigate this instance of the *Factory Method* pattern, the analyst might want to know, for instance, the superclasses of *ET_FileDialog* as well as the methods that are invoked by *MakeFileItem*. The *Context Viewer* will help the analyst answer these questions: window 2 in Figure 26 shows the *Inheritance* view related to *ET_FileDialog.* Furthermore, using the *Design Browser* (window 3), the analyst can retrieve the method *MakeFileItem* and drag and drop it into a new instance of the *Context Viewer* that displays the *Call Actions* view (window 4). The analyst can see in the first place that *MakeFileItem* is invoked by the method *UpdateList*. In addition, the view shows that *MakeFileItem* calls method *Match* in class *Glob*. The analyst might then want to study the definition of the method *Match* in the *Source Code* view (window 5). This provides invaluable context information about *ET_FileDialog* and its role in and around the *Factory Method* pattern instance.

Furthermore, the analyst can investigate the same *e-set* elements in respect to other design patterns. In our example, *ET_FileDialog* is also a participant in instances of the *Template Method* pattern (window 6). The selected instance has four pair of *template methods and hook methods,* as visualized in the *ET_FileDialog* class. This example

presents the case where the design patterns *Factory Method* and *Template Method* are combined to provide a flexible mechanism for object creation in *ET++*.

## Example 3 : Reduction of Cognitive Overhead

In the previous two examples, we saw how visualization in various predefined contexts can enhance software comprehension. Each context view helps the analyst with the construction of a mental model in terms of the user-defined *e-set* elements. Typically, the analyst seeks to verify a hypothesis generated in one view in some other views. This is an example of cross- referencing a mental model. According to Storey *et al.* [34], cross-referencing mental models integrates bottom-up and top down cognitive approaches and improves program comprehension considerably.

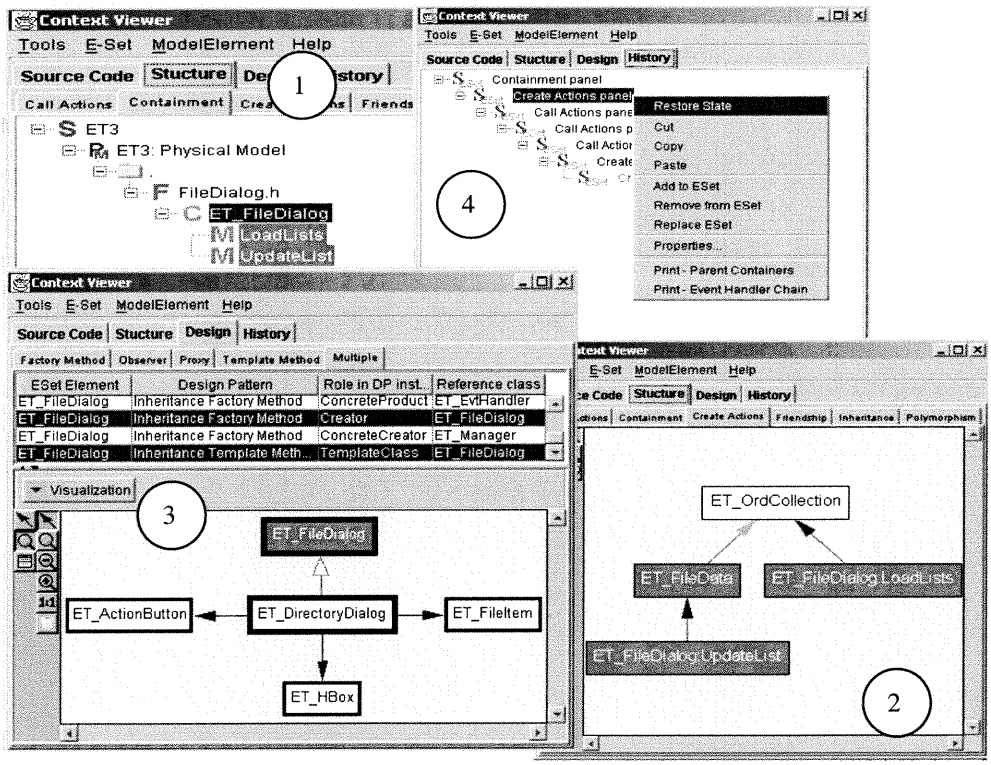


**Figure 27: Reduction of cognitive overhead**

The *synchronization* mechanism of the *Context Viewer* is meant to reduce the analyst's cognitive overhead when cross-referencing mental models. During the investigations

shown in Figure 25 and Figure 26, to switch between the various views at different levels of abstraction, the analyst simply has to click on the tabs that indicate the respective views. For example, it only takes about five clicks to navigate through the views at the structure and source code levels of Figure 25. Obviously, this reduces the user's cognitive overhead and greatly facilitates the cognitive process.

To cut the cognitive overhead even more, the *history* mechanism of the *Context Viewer* may be used. Figure 27 depicts a typical scenario. In this example, the analyst starts the investigation with the *e-set* elements *LoadLists* and *UpdateList* (window 1). Then, after adding *ET_FileData* to the *e-set*, the analyst investigates the create dependency between these elements (*Create Actions* context, window 2). The investigation takes its further course, when the analyst replaces the *e-set* elements by *ET_FileDialog* and considers the *Multiple* view (window 3) to investigate its roles in terms of the *Factory Method* and *Template Method* design patterns.

After a while, however, the analyst might want to resume the investigation of the two initial methods and consider, for example, polymorphism. To this end, the analyst need not invoke the Design Browser or any other SPOOL tool to retrieve the two methods. Instead, the analyst can simply select the *History* panel (window 4) and restore on of the states in which the *e-set* consists of these two methods. Then, the analyst may switch to the desired context views to investigate the two methods. Recall that these switches are very quick because of the synchronization mechanism.

# Chapter 7 : Discussion

In this chapter, we first discuss our approach compared to the comprehension models accepted by the scientific community (see section 2.1). In Section 7.2, we will report on the evaluation of the *Context Viewer* and the SPOOL environment using two requirement lists proposed for software exploration tools [34]. Then, we discuss related work by presenting a feature comparison between SPOOL tools and some representative reverse engineering tools found in both commercial and academic environment (see Section 2.2) in Section 7.3. Finally, we will describe several limits currently related to the SPOOL environment in Section 7.4.

## 7.1 Comprehension Approach

Our comprehension approach, *Visualization in Contexts*, is mainly based on the fact that software elements at low level of abstraction can be visualized in the various contexts at higher levels of abstraction, such as *Inheritance* view at structure level or *Multiple* view at design level, for example. One can thus see this approach as a complement with the various comprehension models presented in Chapter 2.

*Visualization in Contexts* can help the comprehension process in both bottom-up and top-down direction. With this approach, software elements are visualized in various context views at various levels of abstraction and those context views are synchronized. After selected software elements are sent into the *Context Viewer*, a user can make the investigation either from context views at high level of abstraction to context views at low level of abstraction (top-down direction) or opposite (bottom-up direction).

Our approach can also be helpful when a user practices systematic strategies and as-needed strategies for his or her comprehension process. This approach can be used not only when a user tries systematically to understand all the elements and relations of a module or of a subsystem, but also when he or she tries to understand only some parts of the system. In fact, this comprehension approach can support, with various degrees, all the models presented in Chapter 2. One can thus see our approach like an investigation strategy instead of a comprehension model and it can be helpful to those models in terms of reducing cognitive overhead and facilitate comprehension process.

## 7.2 Evaluation of SPOOL Environment

After integrating the *Context Viewer* into the SPOOL environment, we conducted an informal evaluation based on two requirement lists for program comprehension tools known from the literature [34, 28].

In [34], Storey *et al.* present a list of cognitive design elements that should be considered during the development of software exploration tools. The requirements concern support for bottom-up (E1–E3), top-down (E4–E5), and mixed-mode (E6–E7) comprehension, navigation (E8–E9), orientation cues (E10–E12), and disorientation reduction (E13–E14):

E1: Indicate syntactic and semantic relations between software objects.

E2: Reduce the effect of delocalized plans.

E3: Provide abstraction mechanisms.

E4: Support goal-directed, hypothesis-driven comprehension.

E5: Provide overviews of the system architecture at various levels of abstraction.

E6: Support the construction of multiple mental models.

E7: Cross-reference mental models.

E8: Provide directional navigation.

E9: Support arbitrary navigation.

E10: Indicate the maintainer's current focus.

E11: Show the path that led to the current focus.

E12: Indicate options for further exploration.

E13: Reduce additional effort for user-interface adjustment.

E14: Provide effective presentation styles.

The elements E1-E3 and E4-E5 are supported to various extents by the *Visualization in Contexts* approach and the SPOOL environment. Indeed, as described previously, our approach supports both the top-down and bottom-up strategies. The different tools integrated in SPOOL allow for the creation of system models at various levels of abstraction and support the visualization of the relations between software elements.

Elements E8-E9 are mainly supported through the Design Browser, and E6-E7 and E10-E12 mainly through the *Context Viewer*. Specifically, E6-E7 are addressed via the 12 context views and the synchronization mechanism. The *Context Viewer* supports E10-E11 by providing orientation cues and by the recovery of former states via the history mechanism. Regarding E12, a single view or several views combined may well indicate options for further exploration.

As to E13, the multiplication of windows naturally slows down the investigation process. Although SPOOL supports internal windows and thus partially remedies the problem, window management could still be improved in SPOOL. Element E14 is difficult to evaluate in an objective way; yet anecdotal evidence suggests that the adopted presentation styles are indeed useful.

In summary, the tools in the SPOOL environment generally address the suggested elements to a large extent. Yet, further improvement is desirable, in particular in respect to E13.

Singer *et al.* [28] give a list of requirements for software engineering tools that support the so-called *Just-In-Time* comprehension strategy. The list contains functional (F1–F3) and non-functional (NF1–NF7) requirements:

F1: Provide search capabilities such that the user can search for, by exact name or by way of regular expression pattern-matching, any named item or group of named items that are semantically significant in the source code.

F2: Provide capabilities to display all relevant attributes of the items retrieved in requirement F1, and all relationships among the items.

F3: Provide capabilities to keep track of separate searches and problem-solving sessions, and allow the navigation of a persistent history.

NF1: Be able to automatically process a body of source code of very large size, i.e., consisting of at least several millions lines of code.

NF2: Respond to most queries without perceptible delay.

NF3: Process source code in a variety of programming languages.

NF4: Wherever possible, be able to inter-operate with other software engineering tools.

NF5: Permit the independent development of user interfaces (clients).

NF6: Be well integrated and incorporate all frequently used facilities and advantages of tools that software engineers already commonly use.

NF7: Present the user with complete information, in a manner that facilitates the just-in-time comprehension task.

In SPOOL, requirement F1 is supported by the Design Browser. Requirement F2 is addressed by various context views, and F3 is very well supported by the history mechanisms present in the *Context Viewer*, the Design Browser, and other tools.

SPOOL was expressly built to cope with large-scale software systems. It has been used to do researches [20][24] with large-scale software systems, and has shown satisfying performance in those researches. Thus it naturally satisfies NF1. NF2 is also satisfied to a large extent because SPOOL caches *ModelElements* so that the queries usually only take a few seconds [23]. So far, SPOOL processes C++ and Java source code, the main industrial object-oriented languages (NF3). NF7 is well supported, whereas NF4-NF6 are only partially satisfied. Keep in mind, however, that SPOOL still is a research prototype environment.

## 7.3 Related Work

The evaluation presented in the previous section suggests that the SPOOL environment supports quite well the various cognitive aspects related to software comprehension. To be able to better evaluate our environment, we conducted an informal comparison between the context-based tools of SPOOL and the commercial and academic tools described in Chapter 2. We either used evaluation copies of the various tools, or if not available, based our evaluation of the tools' documentation. In what follows, the results of this informal comparison are presented.

The informal comparison is based on two major aspects of our comprehension approach. The first aspect is related to the context views supported by these tools, which promote the construction of mental models by revealing different aspects of the software artifacts' structure and behavior. The second aspect relates to two exploration mechanisms: model cross-referencing is essential to building a mental representation across abstraction levels

and history paths can reduce the exploration effort by giving programmers access to their past investigation path. Table 2 and Table 3 show the results of the comparisons of commercial and academic tools respectively, by giving a weighting (strong (+), medium (+-) and weak (-)) to each criterion and for each tool.

| | | Discover | SNiFF+ | Source-Navigator | Understand for C++ | SPOOL |
|---|---|---|---|---|---|---|
| **Context views** | Source code level | + | + | + | + | + |
| | Structure level | + | - | +- | + | + |
| | Design level | - | - | - | - | + |
| | Customization | +- | - | +- | +- | + |
| **Exploration** | Model cross-reference | - | - | - | - | + |
| | History path | - | - | - | - | + |

**Table 2: Feature comparison of commercial tools and SPOOL**

We conducted an informal comparison between the context-based tools of SPOOL and four commercial tools that we consider most representative for the state-of-the-art, namely, *Discover* [7], *Understand for C++* [39], *SNiFF+* [29], and *Source-Navigator* [31]. Our comparison is based on practical experience with professional licenses of *Discover* and *SNiFF+,* and evaluation licenses of *Understand C++* and *Source-Navigator.* In summary, context views at the source code level are available in all the evaluated commercial tools. For the context views at the structure level, they are supported more or less in all the tools (often accessible via menus) but some object-oriented related views are in general limited. For example, none of these tools supports a context view for polymorphism. On the other hand, none of these tools supports visualization context views at the design level like SPOOL did, which can be very helpful to quickly understand the collaboration of a group of *ModelElements.* Except for SNiFF+, the personalization of context views is supported by these tools, the user can choose one or more than one *ModelElements* (called entity in these tools) in some cases to visualize in views. However, in general they do not allow for the investigation of more than one *ModelElement* at the same time, which limits the information about their interrelationship. Moreover, these tools give very little support to mental model cross-

referencing to help bridging between the various abstraction levels. Finally, no tool provides history mechanism to orient the user in terms of the investigation history path. As a conclusion, none of these tools directly supports our context-based comprehension approach *Visualization in Contexts.*

| | | Rigi | Searchable Bookshelf | SHriMP | SPOOL |
|---|---|---|---|---|---|
| **Context views** | Source code level | + | + | + | + |
| | Structure level | + | + | + | + |
| | Design level | - | - | - | + |
| | Customization | +- | +- | +- | + |
| **Exploration** | Model cross-reference | - | - | - | + |
| | History path | - | - | - | + |

**Table 3: Feature comparison of academical tools and SPOOL**

Many tools have been developed for program comprehension in academia. Some interesting academic tools in respect to context-based comprehension include *Rigi* [45], *SHriMP* [35], and *Searchable Bookshelf* [27]. *Rigi* is a tool allowing for the visualization of diagrams representing the various aspects of a software system (subsystem, files, etc.) in an abstract way. The *Searchable Bookshelf* system comprises advanced capabilities for generating and navigating software structure diagrams (called landscapes). *ShriMP,* finally, is a tool that allows navigation of source code using hyperlinks and that provides some support for context navigation. However, these tools provide little context information at the design level. Moreover, these tools give very little support to mental model cross-referencing to help bridging between the various abstraction levels. Finally, no tool provides history mechanism to orient the user in terms of the investigation history path. As a conclusion, none of these tools directly supports our context-based comprehension approach: *Visualization in Contexts.*

The results of this evaluation show that several aspects of software comprehension supported by the SPOOL environment are poorly supported or absent from the commercial and academic tools that are currently available. Obviously, each of these tools has its strong points for which they were developed and it would be necessary to

include them in our future research in order to introduce their essential points into the SPOOL environment.

## 7.4 Limitations

The advantages of the *Context Viewer* and the SPOOL environment were illustrated by the three examples in Chapter 6 and the evaluation in Section 7.1. In short, together with the rest of the SPOOL environment, the *Context Viewer* can quickly locate the most relevant information for the system elements in terms of different context views. In addition, synchronization and history mechanisms allow the user to bridge different views in an effective way. However, there are several limitations related to the SPOOL environment.

One of these limitations is related to the fact that the SPOOL environment is an environment of investigation and evaluation rather than a development environment. Indeed, the use of SPOOL on a software system requires several stages of preparation before being able for the user to start the investigation. For one thing, the source code of the system must be analyzed, then the information extracted must be imported into the SPOOL repository, which can take several hours depending on the size of the system. At present, the user has to use another development environment (SNiFF+, for example) with the SPOOL environment to obtain the functionality that supports software development.

Another limitations is the fact that the SPOOL environment currently does not provide any mechanism that would store the information and the knowledge obtained in the process of investigation into a repository. Indeed, the backup of the various visualized context views could help the user later and serve as a form of documentation for the newcomer.

# Chapter 8 : Conclusion

Large-scale software system comprehension can be carried out normally in three ways. Unaided browsing is not feasible for a software system with millions of lines of code. Leveraging corporate knowledge and experience can be accomplished with the help of personnel knowledgeable about the subject system, or by resorting to updated documentation of the system. This approach can be very valuable. Unfortunately, experience shows that these knowledgeable experts are not always available and often the system documentation is not up to date. Programmers therefore have a need for techniques and tools that help them find the desired information starting from the source code, which is very often the only available resource.

In this thesis, we presented the *Context Viewer* together with the SPOOL reverse engineering environment to support the comprehension of industrial size, object-oriented software systems. In what follows, we give a more detailed summary and discuss future work.

## 8.1 Summary

The comprehension approach presented in this thesis, namely *Visualization in Contexts,* aims at minimizing the efforts that must be made by a programmer in order to understand a software system. *Visualization in Contexts* mainly depends on three abstraction levels of a software system, which can be extracted in a semi-automatic way: the source code level, the physical and logical structures level, and the design level. Cross-referencing between various preset context views at various abstraction levels is essential if one wants to maximize the comprehension of the system in a reasonable time.

The SPOOL environment was developed to facilitate research in terms of maintenance, design evaluation, and software comprehension. The new tool, *Context Viewer*, was developed by using the SPOOL reverse engineering framework and integrated into the SPOOL environment. The *Context Viewer,* together with other tools in SPOOL, allows supporting the *Visualization in Contexts* comprehension approach. This tool alone contributes to solving many problems related to software comprehension. It addresses the problem of excessive information that is usually extracted from a large-scale software system. With its multiple context views, the *Context Viewer* filters the information and only visualizes what is necessary to the programmer. Moreover, it synchronizes all context views so that various views at different abstraction levels of a system are cross-referenced. Finally, the *Context Viewer* is integrated into the SPOOL environment, and there are interaction mechanisms implemented for communication between them. It leverages the power of navigation and bridges the various levels of abstraction of a system. Therefore, it facilitates the process for programmers to form mental models about the system in hand.

The three example presented in Chapter 6, and the evaluation of the SPOOL environment described in Chapter 7, illustrate how our approach and the usage of the SPOOL environment facilitates software comprehension.

## 8.2 Future Work

To improve the aspects related to the limitations of the SPOOL environment described in Section 7.3, several directions of future works should be undertaken. Indeed, a direction to be explored would be to integrate the functionality of a development environment into the SPOOL environment, so that SPOOL would support both forward and reverse software engineering. This means that SPOOL will have to provide a source code editor, and the information in the repository will be updated according to the changes made in the source code. Moreover, certain mechanisms should be implemented in order to support the storage and analyses of various versions of the software.

Moreover, several additional context views could be added to the *Context Viewer* in the future; for example, the views related to dynamic information about a system and the views related to program slicing or data flow. In addition, certain mechanisms should be implemented and integrated into the SPOOL environment to allow the storage of context views, which serves as documentation for later use.

Finally, the *Context Viewer,* together with other tools (Design Browser, for example), could be useful for other research orientations in the future. For example, these tools could be used for the detection and visualization of the impact of changes and for the inspection of recursive function calls.

# References

Note: All web links mentioned below have been visited in February 2002.


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

[2] Brooks, R., Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies 18*, pages 543-554. 1987.

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

[4] CodeCrawler document. On-line at <http://www.iam.unibe.ch/~lanza/CodeCrawler/codecrawler.html >

[5] daVinci document. On-line at <http://www.tzi.de/daVinci/docs/general_infoF.html>

[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] Finnigan, P., Holt, R., Kalas, I., Kerr, S., Kontogiannis K., Müller, H., Mylopoulos, S., Perelgut, S., Stanley, M., and Wong, K., The Software Bookshelf. *IBM Systems Journal*, Vol. 36, No. 4, pages 564-593. November 1997. On-line at <*http://www-turing.cs.toronto.edu/pbs/papers/bsbuild.html*>.

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

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

[11] GRASP Document. On-line at <http://honiglab.cpmc.columbia.edu/grasp/>

[12] Holt, R., Software Bookshelf: Overview and construction. March 1997. On-line at *<http://www-turing.cs.toronto.edu/pbs/papers/bsbuild.html>*.

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

[14] 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.

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

[16] Letovsky, S., Cognitive processes in program comprehension. *Empirical Studies of Programmers*, pages 58-79, Ablex, Norwood, NJ. 1986.

[17] Linux Cross-Reference, documentation set. On-line at *<http://lxr.linux.no/>*.

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

[19] Robitaille, S. Tool support for understanding industrialsized, 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.

[20] Robitaille, S., Schauer R., and Keller, R.K. Bridging Program Comprehension Tools by Design Navigation. In *Proceedings of the International Conference on Software Maintenance (ICSM'2000)*, pages 22-32, San Jose, CA, October 2000. IEEE.

[21] Yin, R. and Keller, R. K., Program Comprehension by Visualization in Contexts. In Proceedings of the International Conference on Software Maintenance (ICSM'2002), Montreal, Canada, October 2002. To appear.

[22] Schauer, R. and Keller, R. K., The method replacement indicator: A metric for analyzing behavioral substitution. In Proceedings of the International Conference on Software Maintenance (ICSM'2001), pages 754-763, Firenze, Italy, November 2001. IEEE.

[23] Schauer, R., Keller, R. K., Laguë, B., Knapen, G., Robitaille, S., and Saint-Denis, G., The SPOOL design repository: Architecture, schema, and mechanisms. In

Hakan Erdogmus and Oryal Tanir, editors, Advances in Software Engineering. Comprehension, Evaluation, and Evolution, chapter 13, pages 269-294. Springer, 2002.

[24] 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.

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

[26] Shneiderman, B. Software Psychology: Human Factors in Computer and Information Systems. Winthrop Publishers, Inc.,1980.

[27] 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.

[28] 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.

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

[30] Soloway, E., Pinto, J., Letovsky, S., Littman, D. and Lampert, R. Designing documentation to compensate for delocalized plans. Communications of the ACM, Volume 31 , Issue 11, pages 1259-1267. 1988.

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

[32] S. R. Tilley, The canonical activities of reverse engineering. Baltzer Science Publishers, The Netherlands, February 2000.

[33] Storey, M.-A. D., A Cognitive Framework For Describing and Evaluating Software Exploration Tools. PhD Thesis, Technical Report, School of Computing Science, Simon Fraser University. December 1998.

[34] 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.

[35] 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.

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

[37] Tom Sawyer Graph Layout Toolkit document. On-line at < http://www.tomsawyer.com/glt/index.html >

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

[39] Understand for C++ Documentation set. On-line at *<http://www.scitools.com/ucpp.html>*

[40] VCG Overview. On-line at < http://rw4.cs.uni-sb.de/~sander/html/gsvcg1.html >

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

[42] Visualizing Graphs with Java documentation. On-line at <http://www.eng.auburn.edu/department/cse/research/graph_drawing/graph_drawing.html >

[43] Von Mayrhauser, A. and Vans, A. M., Program comprehension during software maintenance and evolution, IEEE Computer (Vol. 28, No. 8), pages 44-55. 1995.

[44] Von Mayrhauser, A. and Vans, A. M., Comprehension processes during large-scale maintenance. In *Proceedings of the International Conference on Software Engineering (ICSE'94)*, pages 39-48, Sorrento, Italy. May 1994.

[45] Wong, K. and Müller, H., Rigi User's Manual—Version 5.4.4, University of Victoria, Victoria, Canada, June 1998. On-line at *<ftp://ftp.rigi.csc.uvic.ca/pub/rigi/doc/rigi-5.4.4-manual.pdf>*.

# Appendix : Specification of Context Viewer