Navigation in Object-Oriented Reverse Engineering

Diplomarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Daniel Schweizer

2002

Leiter der Arbeit:

Prof. Dr. Oscar Nierstrasz
Dr. Stéphane Ducasse
Michele Lanza

Institut für Informatik und angewandte Mathematik

Further information about this work, the used tools and an *online* version of this document can be found at:
http://www.iam.unibe.ch/~scg/

The address of the author:

Daniel Schweizer
Effingerstrasse 9
CH-3011 Bern

or

Software Composition Group
University of Bern
Institute of Computer Science and Applied Mathematics
Neubrückstrasse 10
CH-3012 Bern
dschwzr@iam.unibe.ch
http://www.iam.unibe.ch/~dschwzr/

Abstract

Tool support is needed to cope with the complexity and the large amounts of information in reverse engineering. By creating representations in another form, often at a higher level of abstraction, state-of-the-art tools aid in reducing complexity and gaining insights into parts of a system's structure. However, orientation and navigation among these representations remains difficult. Often superfluously tool-induced effort is needed to perform a certain task. We call this artificially added effort *friction*.

Tools with the right navigation support can reduce this *friction*, and increase productivity. This work classifies navigation in models of object-oriented software systems, and shows that among the great number of possibilities, only a few make sense. We determine which kinds of navigation steps are useful, and why. We summarize our experience and best practices of state-of-the-art tools in a set of requirements for an ideal reverse engineering tool.

As a validation for these requirements, we analyze data about the user's behavior during reverse engineering sessions. To collect that data, and for studying various ways of navigation and orientation, we built *MooseNavigator*, a prototype reverse engineering navigator.

Contents

Abstract **i**

1 Introduction **1**
1.1 Friction in Reverse Engineering 4
1.2 Reducing Friction with Navigation 6
1.3 The Structure of this Document 7

2 Background **8**
2.1 Knowledge Management 8
2.1.1 Managing Information 9
2.1.2 Topic Maps 13
2.1.3 Navigating Information 17
2.2 The FAMIX Reverse Engineering Model 19
2.3 Usability 21
2.4 Concerns Identified 24

3 Context & Requirements **27**
3.1 Moose & CodeCrawler 27
3.2 The SORTIE Experience 28
3.2.1 Experience 28
3.2.2 Results 32
3.3 Tool Requirements 33

4 Classifying Navigation **40**
4.1 Navigation Steps between Tool States 41
4.2 Semantic Navigation Steps 42
4.3 The Dimensions of Navigation 45
4.4 State-of-the-Art Navigation 45

5 Navigation for Reverse Engineering Tools **47**
5.1 Perceptions 47
5.2 A Prototype - MooseNavigator 50
5.2.1 Conception 50
5.2.2 User Interface 51

5.2.3 Metrics . . . 55
5.2.4 Layouts & Views . . . 56
5.2.5 Reports . . . 58

6 Experiments **59**
6.1 Automatic Navigation Support . . . 59
6.2 Experiment with Students . . . 63
6.2.1 Observations . . . 63
6.2.2 Experience . . . 65
6.2.3 Results . . . 66
6.3 Experiment with an Expert . . . 66
6.3.1 Observations . . . 67
6.3.2 Experience . . . 69
6.3.3 Results . . . 70
6.4 Summary . . . 71

7 Conclusion **72**
7.1 Summary . . . 72
7.2 Main Contribution . . . 73
7.3 Outlook & Future Work . . . 74

A Moose **75**
A.1 Meta Model . . . 76
A.2 CodeCrawler . . . 77
A.3 MooseExplorer . . . 77
A.4 MooseFinder . . . 78

B SORTIE Report **81**
B.1 Project Background . . . 81
B.2 Project Success . . . 82
B.3 SCG Report . . . 84

C State-of-the-Art Tools **95**
C.1 Introduction . . . 95
C.1.1 Selection . . . 95
C.1.2 Scope . . . 98
C.1.3 Template . . . 99
C.2 Tools . . . 100
C.2.1 Eclipse . . . 100
C.2.2 Javadoc . . . 104
C.2.3 Rigi . . . 109
C.2.4 SHriMP . . . 113
C.2.5 Small Worlds . . . 118
C.2.6 TheBrain . . . 122

C.2.7 Together 126
C.2.8 Additional Features 131
C.3 Summary 132

D MooseNavigator Implementation 134

List of Figures

1.1 Software Reengineering Lifecycle. 2

2.1 The Elements of the FAMIX Model. 20
2.2 The Entities in the FAMIX model. 20

3.1 Screen Capture of *CodeCrawler*. 28
3.2 SORTIE System Complexity. 29
3.3 SORTIE Class Blueprint. 30
3.4 SORTIE System Packages. 31

5.1 Screen Capture of *MooseNavigator*, Tools In-lined. 51
5.2 Screen Capture of *MooseNavigator* Session Viewer. 52
5.3 Screen Capture of *MooseNavigator* Description Viewer. 53
5.4 Screen Capture of *MooseNavigator* System Overview. 54
5.5 Screen Capture of *MooseNavigator* Filter Editor. 54
5.6 Screen Capture of *MooseNavigator* Filter Library Editor. 55
5.7 A Typical Circle (Double) View. 56
5.8 Typical Navigation Trails Evolving over Time. 57

6.1 Procedure of a Typical Student User Session. 63
6.2 Distribution of Selected Views by Students. 64
6.3 Percentage of Visited Classes and Entities by Students. 65
6.4 Procedure of an Expert User Session. 67
6.5 Distribution of Selected Views by an Expert. 68
6.6 Percentage of Visited Classes and Entities by an Expert. 69
6.7 Navigation Trails in the Duploc Experiment. 70

A.1 *Moose* Architecture. 76
A.2 FAMIX Core. 77
A.3 Screen Capture of *CodeCrawler*. 78
A.4 Screen Capture of *MooseExplorer*. 79
A.5 Screen Capture of *MooseFinder*. 80
A.6 Screen Capture of *MooseFinder* Query Composer. 80

C.1 Screen Capture of Eclipse. 100

C.2 Screen Capture of Internet Explorer with Google Toolbar & Javadoc. 104
C.3 Screen Capture of Rigi. 109
C.4 Screen Capture of SHriMP. 113
C.5 Screen Capture of Small Worlds. 118
C.6 Screen Capture of PersonalBrain. 122
C.7 Screen Capture of Together ControlCenter. 126

List of Tables

2.1 Tasks of Knowledge Management. 11
2.2 Acceptance Factors of Knowledge Management Tools. 12
2.3 Relations in Class Hierarchies. 16
2.4 Mapping between FAMIX Elements and its Graphical Representations. 21
2.5 Concerns of Designing a Reverse Engineering Navigator. 25

4.1 Navigation between Tool States. 41
4.2 Navigation Steps Starting from a Class. 43
4.3 Navigation Steps Starting from an Attribute. 44
4.4 Navigation Steps Starting from a Method. 44
4.5 State-of-the-Art Navigation Overview. 46

6.1 Methods of MSEAttribute's Fullclass. 60
6.2 Attributes of MSEAttribute's Fullclass. 61
6.3 Average Time between Student User Interactions. 64
6.4 Average Time between Expert User Interactions. 68

B.1 SORTIE Participating Tool Teams. 82

C.1 Tools of the Category Knowledge Management. 96
C.2 Tools of the Category Text Browsing. 96
C.3 Tools of the Category Source Code Browsing. 97
C.4 Tools of the Category IDE. 98
C.5 Tools of the Category UML Modeling. 98
C.6 Tools of the Category Static Analysis / Documentation. 98
C.7 State-of-the-Art Features Overview. 133

Chapter 1

Introduction

> *"A successful and used software product must be subject to evolution, else it becomes progressively less satisfactory."*
>
> -Manny M. Lehman [LEHM 96]

This work is about navigation in object-oriented reverse engineering. In this first chapter we explain why this is an issue and define a terminology for reverse engineering. We detect the key objectives of reverse engineering and identify problems that make orientation and navigation in representations of software systems difficult. In a further section we introduce the concept of *friction* which is the superfluously tool-induced effort needed to perform a certain task. At the end of this chapter you find an overview of the structure of this document.

Today's continuously changing environments influence the methods of developing and maintaining software. Several factors make object-oriented re- and reverse engineering an important chapter of software development. The trend towards bigger systems, component- and framework architectures and the dependency to development kits, huge class libraries, and other third party components makes it difficult to keep the complete overview of a complete software system. Programmers do not like to document what they implement, and the job fluctuation is high. As a result, we find many not- or poorly documented systems [Duc 99] [Dem 02]. In combination with the increasing popularity of the object-oriented paradigm, this situation causes people to rethink traditional software development, and results in iterative processes. For example, an agile process like *Extreme Programming* [BECK 00] [JEFF 01] emphasizes short release cycles, collective code ownership, merciless refactoring and rotating developer pairs.

Iterative processes turn software development into a continuous reengineering or maintenance job. Typically the system's maintainers are not its original designers. In this work we try to identify ways to support developers in program understanding. To avoid confusion, we begin by introducing a vocabulary for the

common understanding of terms and concepts. After discussing the *software reengineering lifecycle*, we continue by defining *friction in reverse engineering*. We list the *friction*'s major drivers, and present our strategy for reducing it.

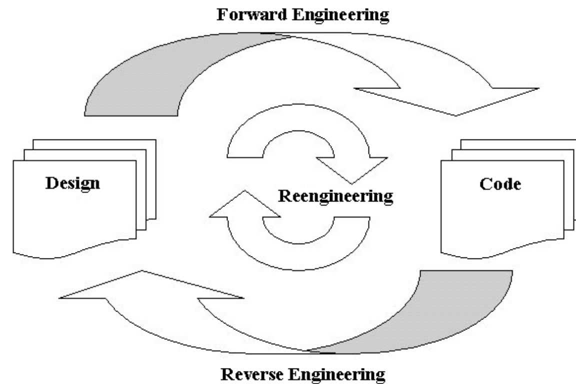


Figure 1.1: Software Reengineering Lifecycle.

The elements of the *software reengineering lifecycle* are *Reengineering*, *Forward Engineering*, and *Reverse Engineering*. Figure 1.1 illustrates how the elements interrelate. The following definitions are in compliance with Chikofsky [CHIK 90]:

> *"**Reengineering** is the examination and alteration of a system to correct faults, improve the design or performance, and change the product to meet additional or changing requirements"*

> *"**Forward engineering** is the traditional process of moving from high-level abstractions and logical implementation-independent designs to the physical implementation of a system"*

> *"**Reverse engineering** is the process of analyzing a system to identify its components and interrelationships in order to create representations in another form, often at a higher level of abstraction."*

Before a developer is able to make modifications in the traditional forward engineering view, he has to spend some effort in studying and trying to understand the structure and architecture of the legacy system. This task of reverse engineering is also called design recovery. If we want to support the reverse engineers at their work, we must first know in detail about the versatile list of their objectives. Here is the list of objectives according to Chikofsky [CHIK 90], each complemented with a brief explanation.

- Cope with complexity. It is difficult to keep the overview of large amounts of source code. One way to cope with complexity, is reducing the volume

in hiding some details, and stressing some aspects. Other ways include the creation of different meaningful representations.

- Generate multiple views. Depending on your current point of view, alternative ways of putting things together may be appropriate - new layouts and views are required.

- Recover lost information. This includes the re-documentation of a system, if there is no documentation. Often the existing documentation is based on the original design, and it is not in sync with the current system anymore. Sometimes companies recover also forgotten details on their processes and workflows from within software implementations.

- Detect side effects. Without a clear picture of the overall architecture, and without the support of analysis tools, it is often hard to identify interfaces and dependencies. Especially if you make modifications to software, where many technologies from different manufacturers must work together, it is hard to determine the impact to the total system.

- Synthesize higher level abstractions. Higher level abstractions can evolve during the reverse engineering process, with the input of additional concepts and knowledge of users. An example is the aggregation of artifacts to containers, by inspecting the "virtual name spaces" - implicit semantics, derived from a assumed naming scheme.

- Facilitate reuse. Developers avoid reusing what they do not understand. Higher level abstractions help in reducing the efforts needed for comprehending the design of third-party components.

The primary motivation for the objectives described above is normally doing modifications, which implies reverse engineering as an inevitable part at the beginning of the reengineering task. To keep the system in a good shape, alterations are frequently accompanied by efforts of restructuring. Restructuring is today, especially in the context of object technology, more popular under the name *refactoring*.

> *"**Refactoring** is the inner modification of a system in favor of a better design, without changing its external behavior."*
>
> -Martin Fowler [FOWL 99]

An improvement of the design leads also to systems that are easier to understand, maintain and extend. Sometimes performance or security issues motivate the refactoring of a system. More candidate refactorings can be detected via anomalies [RIEL 96] and hotspots. It can be wise to think about whether **foregoing** *refactorings* simplify alterations, and it is a good piece of advice, to check if the design

needs *refactoring* **after** having done a modification.

While the field of design patterns is widely explored and documented in literature [GAMM 95] [BECK 97] [ALPE 98] - up to now, approaches towards a methodology for the better understanding of object-oriented systems, are in its beginnings [DUCA 01a] [Dem 02] [STOR 97]. We believe that a key prerequisite of successful reverse engineering is to have a clear plan. First of all, you should think about what you want to find out. Once you have defined the concrete goals, you need good tools supporting you in exploring the system efficiently and in finding and remembering what you were looking for. To determine decisive factors for efficiency we first identify typical sources of inefficiency. This leads us to the concept of *"friction"* in reverse engineering. In the following we explain what we understand by this term and why we should try to reduce it.

1.1 Friction in Reverse Engineering

Tool support is needed to cope with the large amounts of complex information while reverse engineering a system. State-of-the-art reverse engineering tools aid in reducing complexity and gaining insights into parts of a system's structure by creating representations in another form, often at a higher level of abstraction, however orientation and navigation among these rather static representations is difficult. The exploration of large systems offers a wide range of ways to get lost [CONK 87] [NIEL 90] [MCKN 91]. On the way to the target you lose a lot of time in not being able to directly get where you want. We call this tool-induced loss of productivity *"friction"*:

> **Definition:** ***Friction in reverse engineering*** is all the additional, artificially tool-induced effort, which is superfluous but necessary to perform a certain action for achieving a certain goal.

In terms of Human Computer Interaction (HCI), *friction* is mainly caused by the design of inefficient Graphical User Interfaces (GUIs). Another term for *friction*, although exclusively in respect to HCI, is *excision* [COOP 95]. State-of-the-art tools help in reducing the complexity in reverse engineering object-oriented systems, still they induce a considerable amount of *friction*. Here are the six major groups of contributors which we compiled from existing literature [COOP 95] [RASK 00] [WARE 00]: *Incompleteness of the model and features, indirection, oversaturation, red herrings, degradation of knowledge,* and *lack of classification*. We describe them below in more detail:

1. **Incompleteness of the model and features**. For being able to find information, a first requirement has to be fulfilled, before all others: Completeness of the model and features. This is the prerequisite to navigate along real references or interrelationships. We cannot get there if it is not there or there

is no feature for getting there. It is also important that all information can be accessed in the same tool. Switching over to other applications to find missing pieces of information means great friction.

2. **Indirection**. Sometimes we know exactly where we want to go. But it takes us many interactions to get there. Not being able to access relevant information directly is expensive, tedious and de-motivating, especially when we perform the same task many times. Support for getting fast from A to B provides shortcuts and includes the elimination of obstacles.

3. **Oversaturation**. We are bad in coping with large numbers of entities simultaneously and sometimes we get lost in the great number of entities and possible navigation paths. A way of reducing complexity (and the chance of getting lost) is by reducing the amount of information that is concurrently presented, for example by applying filters.

4. **Red herrings**. We easily get lost, being attracted by a red herring which is something of which we are mistaken by what it is or by its contributing value to our current task [KICZ 96]. Overloaded menus with many seldom used commands distract the user. A well designed tool reduces the possibilities to a small number of navigation steps, since often the useful actions in a certain situation are few and always the same [GRIS 92]. Straightforward dynamic menus make the choice for further actions simpler. The automatic and immediate display of all relevant information about the currently selected entity makes additional interactions for visiting detailed information superfluous. A history function is needed to provide ways back out of blind alleys in the labyrinth.

5. **Degradation of knowledge**. To support reuse of perceptions a system must remember what we find out once, and we must be able to access this knowledge in the future. We need some sort of memory to reduce the degradation of discovered knowledge. In state-of-the-art reverse engineering tools, bookmarks or similar capabilities are rare. Most reverse engineering sessions do not result in anything storable, except static reports or pictures. Users throw away everything after every single use. Users must restart from scratch when they want to find out the same thing at a later moment of time. Throwing away instantly the insights is inefficient. After all, users should be able to generate detailed reports about their sessions, these reports must be adaptable and extensible. The result sets must be in a form that can be imported for further use in other data analysis tools. This is even more important, when reverse engineering in a team where support for sharing and exchanging sessions, resources, and insights with colleagues is needed.

6. **Lack of classification**. Without a concept of classification, it is hard to distinguish different kinds of entities of information. Since normally not every entity is of the same importance, it can be helpful to rate them according to

their relevance, to a certain aspect, or a context. Usually you have no idea where you already passed by and which area of the system you or anybody else did not yet touch. The coverage of visited parts of a system, can also be expressed by classification.

1.2 Reducing Friction with Navigation

"I saw a statistic from a U.C. Berkeley study saying that it took us 300,000 years to generate an amount of information and will take only the next 2.5 years to get the same amount."

-Thomas H. Davenport [DAVE 98]

One of the primary goals of software reverse engineering is very much the same as in knowledge management: It is finding relevant information. The goal of this document is to present successful navigation strategies for reverse engineering. For that, we start by identifying the concerns to be considered in building navigation tools. This set of concerns is used to structure the discussion of best practices and example solutions of state-of-the-art tools from knowledge management and reverse engineering.

We identify appropriate navigation features to be one possible force to considerably reduce friction. We discuss and classify possible navigation steps on models of object-oriented software systems in order to determine useful and useless navigations. We study possibilities for reducing the chance of getting lost as well as to support learning from previous investigations.

The statements are validated by the analysis of data which was collected to study the behavior of users during reverse engineering sessions. Indeed, the location of so called trails, or so far untried areas of a system - generally, meta information about the popularity and experiences of past investigations - can help to build more efficient navigating tools.

As a major result of this work we collect a set of requirements for an ideal reverse engineering tool that supports orientation, navigation, as well as learning from previous investigations. We believe that this can further reduce complexity and thus increase productivity in reverse engineering.

1.3 The Structure of this Document

The rest of this document is organized as follows:

- The background. Chapter 2 starts by encompassing the broader context of navigation in reverse engineering: namely managing, modeling and navigating information. The chapter presents our reverse engineering meta model, identifies some issues of usability, and results in a set of concerns that has to be addressed when thinking about the ideal navigation support of reverse engineering tools.

- The motivation. We present the strengths and weaknesses of our reengineering environment *Moose* and the experiences we made in an industrial case study in Chapter 3. This leads us to a set of concrete tool requirements.

- A classification. Chapter 4 introduces some terms and concepts for distinguishing different kinds and the building blocks of navigation. We summarize state-of-the-art solutions for efficient navigation in reverse engineering tools and identify some gaps.

- Our contribution. We describe the main perceptions gained in this work in Chapter 5. We present our prototype *MooseNavigator* and give some examples of how the tool requirements could be implemented in our environment.

- The validation. In Chapter 6 we first describe an experiment of searching ways to dynamically provide features for navigation. After that we describe and analyze data about the navigation behavior of reverse engineers. The data was collected in two additional experiments of using our prototype.

- The conclusion. We summarize what we have done and give an outlook to future work in Chapter 7.

- In addition there are four appendices: Appendix A presents the *Moose* reengineering platform and its tools that were developed in the Software Composition Group at the University of Bern.

- While we describe our experience made in the SORTIE case study in Chapter 3, we summarize the project background and success in Appendix B. There you can also find the originally submitted SCG project report.

- In Section 4.4 we present a summary of state-of-the-art tools with an exclusive focus on navigation and orientation. The complete evaluation of the tools can be found in Appendix C.

- We describe the implementation details of our prototype *MooseNavigator* in Appendix D.

Chapter 2

Background

"Jede Erinnerung schreibt die Vergangenheit neu."[1]

-Gilbert Probst [PROB 99]

In this work, we are interested in navigating information for reverse engineering. Navigating information is not new - the field of information retrieval, today rather known as knowledge management, studies this for decades. Before we focus on navigation in reverse engineering, we consequently introduce general knowledge management practices and terminology. For the following discussion we split the field of knowledge management into the three subsections managing-, modeling-, and navigating information. In the section about modeling information we present a concrete example model. It is the topic maps model. The next section presents the FAMIX reverse engineering model which is the base of our research and reengineering environment. After that follows a brief survey on usability. The chapter ends with an accumulated list of concerns for the design of navigation tools. This list of concerns is later used as a structure to analyze the problems in navigation and discuss state-of-the-art solutions.

2.1 Knowledge Management

"Sharing knowledge means multiplying knowledge."

-Thomas H. Davenport [DAVE 98]

The goal of this section is to introduce the concepts and a terminology for managing, modeling and navigation information in knowledge management. Many of the concepts and a considerable amount of the terminology can be reused in reverse engineering.

[1] Literally: "Every reminiscence rewrites the past."

Sharing knowledge means multiplying knowledge. This substantial difference between knowledge and any other physical good, is one driver of the information flood, with which we are faced in knowledge management. Many criteria to measure the usefulness of tools and methods in knowledge management, are the same as in reverse engineering. Indeed, some of the primary goals are to support human thinking and coping with huge and complex sets of information, to filter them, and to find relevant pieces, with regard to your current context.

2.1.1 Managing Information

This subsection guides you to the background of information and knowledge management. It defines what we understand by knowledge and classifies different kinds of knowledge. We also present a terminology for the methods and the tools of knowledge management.

Background

The management of information and knowledge evolved to one of the most important disciplines in today's world. The global knowledge economy is predicted to be the successor to the information age. We all use tools to improve our capabilities, in most fields of our lives, but usually not in thinking - up to now. As a result of the increasing needs for efficiency in the age of globalization the process of knowledge creation and transformation is growing to be a key competence of any business [RHEI 85]. An example proving the monetary interest, not only on what the company owns, but also what it knows is SKANDIA's idea of "Intellectual Capital" recently applied also at ERICSSON [2]. At the end of the year SKANDIA's shareholders are not only presented a balance sheet with a financial statement but also the intellectual capital, as a combination of customer human capital and organizational capital.

Like many technology fields, the area of knowledge technology has suffered from overly high expectations and excessive levels of hype, particularly with regard to expert systems. It is fair to say that many of the *"new"* technologies and tools of knowledge management date one or two decades back. Those days they were called information systems. The discipline was also known as *Information Retrieval*.

Knowledge Classification

Scientists could not yet agree on a unified definition for knowledge. One reason for this is the continuous confusion about the terms *Data*, *Information* and *Knowledge*. Here is how we use them: *Symbols* are a subset of *Data*. A *Symbol* is taken out of

[2] More on Intellectual Capital at SKANDIA or ERICSSON at: http://www.skandia.com/ and http://www.ericsson.se/intellectualcapital/

an *Alphabet* to build a word. A word is an instance of *Data*. To construct words you need a certain syntax or language. Within a specific context one or more words can become *Information*. Only combined and interpreted *Information* will ever have the chance to become *Knowledge*. Here is our definition for knowledge:

> *"Knowledge is the ability people have to use information to solve complex problems and adapt to change; the individual ability to master the unknown; the ability to act."*
>
> -Karl-Erik Sveiby [SVEI 97]

We distinguish two kinds of knowledge:

1. Tacit knowledge. Tacit knowledge is highly personal and hard to formalize and communicate - thus not directly accessible to others. It consists of know how and mental models, beliefs and perspectives based on experience. It can be transformed to explicit knowledge - e.g., by socializing.
2. Explicit knowledge. Explicit knowledge is formal and systematic, can be stored and easily be communicated and shared. It is articulated - the words we speak, the books we read, the reports we write, the data we compile.

Tacit knowledge exists in human brains only, and can therefore not be stored on any other media. The only thing that can be put into a non-human system is data. If the knowledge contributor and the knowledge customer have the same language and the same cultural background, often this data is easily converted to information. After that, you need to have the know-how to present this information in such a comprehensive way - that hopefully people can regenerate knowledge out of it. However, we sometimes also use knowledge as synonym for information - for simplification and compatibility with terminology in common literature. Patents, trademarks, copyrights, and trade secrets appear to be viable candidates for explicit knowledge, and of course software.

Knowledge Management Methods

The six core tasks of knowledge management by Probst [PROB 99] are: Identify, Store, Utilize, Distribute, Create (internal), and Buy (external). Table 2.1 shows our **extended** list of tasks which also takes in account slightly varying definitions found in literature [DAVE 98] [ANTO 99].

Studies have shown that companies primarily collect and manage knowledge about methods rather than about products, customers or market and competitors. But finally, the key success factors for the implementation of knowledge management in an organization, are rather soft: Culture and Vision of the Company,

Task	*Comment*
Identify / detect knowledge	Identifying knowledge carriers, - flow, - structures, and generating a knowledge portfolio is the first step towards knowledge management. It is important to detect also the knowledge gaps. Benchmarking tells you what you know in comparison to others.
Create knowledge (internal / external)	Enlarging the total sum of knowledge can be achieved by techniques and measures like education, think tanks, forums, insourcing, outsourcing, or hiring external experts and consultants.
Collect knowledge	Tacit knowledge must be transformed into explicit knowledge, for it can be stored in databases, dictionaries, best practices, red books, white papers, or other sorts of containers. A weak form of collecting knowledge is to collect information about knowledge carriers or places where further information can be found. This leads to white pages (people), yellow pages (organizations), or blue pages (governmental departments).
Filter / evaluate knowledge	While collecting information, you should also rate it. Criteria can be relevance, trustworthiness of information source, timeliness, scope. This task is complex and needs experts.
Categorize, synthesize and structure knowledge	For simpler management and easier relocation of information, as well as to know the scope of specific pieces of knowledge, you need to categorize it. To be able to perform this task, the antecedent formation of a controlled vocabulary and a common ontology is necessary.
Store knowledge	Media, that theoretically can store knowledge or information are: DNA, brains, software, hardware, and books. Today's knowledge economy attempts to port most of the information to be managed in hard- and software.
Distribute and share knowledge	"Lessons Learned" represent a popular method to get better in recurring tasks. They not only show how to do something, but also how not to do. *Push* means distributing knowledge actively, while *pull* means offering ways for people to come and get it.
Utilize knowledge	Apply the gathered knowledge. This is the first task where you can make profit of the preceding tasks of preparation.
Update and maintain knowledge	Without a continuous feedback, reviews, and contributions of knowledge-workers; without the cooperation between end-users and operators, knowledge management can not be kept alive. The process of making explicit to others what you learn, is called "unlearning".

Table 2.1: Tasks of Knowledge Management.

Structure and Processes, Employees Motivation and Qualification, Encouragement through Top Management, Pressure to Succeed, Clear Targets, Training and Education, Incentives, and Integration of External Knowledge [Anto 99].

The majority of these soft factors can be disregarded in the context of reverse engineering, because a lot of work in the non-technical tasks of identifying, creating, and collecting knowledge is done by the computer that parses the source code and generates the models automatically. This is an advantage, since with that, the work is less vulnerable to lazily contributing knowledge-workers, political games or other barriers.

Knowledge Management Tools

The success factors for implementing knowledge management tools in organizations include simplicity, efficiency, and still more barriers (Table 2.2).

Factor	*Comments*
Simplicity	Tools must not be complicated to operate, otherwise users do not use them. The user must see a clear benefit of using a tool, over traditional ways of searching for information.
Maintainability	A knowledge tool must be always up-to-date, otherwise people will stop trusting it. Easy contribution must be possible, otherwise users contribute rarely to enlarge or improve the knowledge pool.
User-Friendliness	The tool must not only be simple to use, but also provide nicely presented results. This data must be available for further processing. Queries and result sets must be storable.
Political Barriers	Sharing knowledge means giving it away - this often causes the fear to lose power or the "right to exist". This benchmark thinking within the team can hinder people to contribute.
Psychological Barriers	No habit in working with tools frightens people to harm any content. Missing self confidence and the fact that everybody will be able to criticize contributions lets many users keep their insights for themselves. Other people could maybe interpret contributing to the knowledge pool as having nothing more important to do. The "not-invented-here syndrome" can decrease the acceptance of external knowledge.
Structural Barriers	"Internal intransparency" and unprecise responsibilities cause knowledge management to be no "real project" on which working hours can be booked on - this leaves the impression that it is not that important.

Table 2.2: Acceptance Factors of Knowledge Management Tools.

Knowledge management tools can roughly be separated in two groups: Tools for supporting communication and tools that provide access to information.

1. Communication support: Communication is important for sharing knowledge. Communication can be supported by technology: E-mail, chat, instant messaging, electronic blackboard, wiki and more sophisticated knowledge systems. A minimal technical communication support is necessary but not sufficient. It cannot replace physical meetings and oral communication [PROB 99]. However technology should enable work everywhere and sharing information transparently. Another way of making people communicate are institutions like a hotline, forums or knowledge brokers. However still more important than technology, is the company's culture and organizational methods; clear targets, training and education, pressure to succeed, or incentives.

2. Information access: This group of tools includes libraries and archives that primarily contain textual documents or databases. Especially popular in consulting companies are "Lessons Learned" and "Best Practices". They are what design and process patterns are to software development and what we

expect from reverse engineering patterns. 'Yellow Pages" listing experts, kind of "who-is-who" sometimes can be derived from implicit information in documentation or annotations of the version control system. Other libraries and archives include search engines, meta search engines, indices, dictionaries, encyclopedias and product catalogues. Moderated libraries include directories or special databases for genealogy or patents, enhanced content management systems, and information portals. Information maps build a common way for visualizing knowledge. Whether landscape maps, hyperbolic trees or simple hyperlink collections, the content can vary independently of the type; they can show knowledge carriers, knowledge portfolios, knowledge configuration, knowledge implementation, knowledge flow, or implementations of local theories. When coping with models of object-oriented code, trees and graphs are used to represent the artifacts and relationships.

3. Artificial intelligence: Neural networks and other technologies were used to build *expert systems*, which try to support making decisions by deriving advice from a set of rules. Associative search agents try to inform you about things from which they reason that it would be relevant to you.

Being sure that two people are talking about the same thing, requires a common thesaurus of technical terminology and a controlled vocabulary - otherwise tools will fail. A model for topics and aims is needed. In the following we present ways of modelling and navigating knowledge. We show that the way we model our information space, has an impact on the possible ways of managing and navigating them later.

2.1.2 Topic Maps

Within the fields of semantics and artificial intelligence, many concepts were already - and still are - used to describe various models for representing knowledge structures within a computer. These include: "Semantic Networks" [GRIF 82] [LEHM 92], "Semantic Web" [BERN 99], "Associative Networks" [FIND 79] and "Resource Description Frameworks (RDF)"[3], the "Knowledge Interchange Format (KIF)"[4] or "Conceptual Graphs" [YANG 93]. Many of these already correspond closely to the topic/association model. The concept of topic maps, by adding the topic / occurrence axis, provides a means for "bridging the gap" between knowledge representation and the field of information management.

Because of the general nature of topic maps, we chose to use them to introduce the concepts for modeling information. By doing that, we visit the relevant issues, and typical classes of topic types and relations. This helps also in understanding

[3] http://www.w3.org/RDF/
[4] http://logic.stanford.edu/kif/

the possibilities of navigation in representations of object-oriented code, and builds a viable vocabulary for further discussions.

The topic map standard defines the model and interchange syntax for topic maps [ISO 99]. This is not the only way of representing knowledge or information, however it covers the centrals issues, and is a perfect example for discussing the questions relevant to our context. The original motivation for topic maps dates back to the early 1990's related to the desire to model intelligent electronic indexes in order to be able to merge them automatically. Today topic maps are considered to become the "GPS of the information universe" [Rath 99]. Topic maps became an ISO 13250 industry standard in 1999[5].

Topic Map Concepts

We start with a short introduction to the key elements of the topic map model, which are: *Topic, occurrence, association, scope* and *facet*, as well as its corresponding *roles* and *types*. For each concept, we give one or more examples.

Topic. Topics are the building blocks of a topic map. A topic, in its most generic sense, can be anything, regardless of whether it exists or has any other specific characteristics. An individual topic is an instance of zero or more *topic types*. Topic types themselves are defined as topics again. Topics have three kinds of characteristics: *names, occurrences,* and *roles in associations*. Example topics are *Bern, Switzerland*, and *University of Bern*.

Occurrence. A topic may be linked to one or more information resources that are relevant to the topic in some way. Such resources are called occurrences of the topic. Occurrences may be of any number of different types. Such distinctions are supported in the standard by the concept of the *occurrence role*. As with topic types, occurrence roles are topics. Occurrences of *Bern* could be a web site with tourist information or a city map.

Association. A topic association is (formally) a link element that asserts a relationship between two or more topics. Just as topics can be classified according to their type (class, method, attribute, etc.) and occurrences according to role ("Access", "InheritanceDefinition", etc.), so can associations between topics be classified according to their type. Viable candidates for a "defines" are "belongsTo" (structural information), "invokes", "accesses", and "accessed_by". As with most other constructs in the topic map standard, association types are themselves defined in terms of topics. The ability to do typing of topic associations increases the expressive power of the topic map, making it possible to group together the set of

[5]More information about topic maps can be found online. A recommended starting point is http://easytopicmaps.com/

topics that have the same relationship to any given topic. This is of great importance in providing intuitive and user-friendly interfaces for navigating large pools of information. Each topic that participates in an association has a corresponding "association role" which states the role played by the topic in the association. Also association roles are regarded as topics in the topic map standard. An example association "is_in" connects the two topics *Bern* and *Switzerland*. This association has the type *Location*.

Scope. From the preceding discussion we see that topics can have various characteristics assigned to them: they can have names, they might have occurrences, and for every association in which they partake, they have a role. These different kinds of assertions that can be made about a topic are collectively known as "topic characteristics". In the topic map standard, any assignment of a characteristic to a topic, be it a name, an occurrence or a role, is considered to be valid within certain limits, which may or may not be specified explicitly. The limit of validity of such an assignment is called its scope, also scope is defined in terms of topics. If not further specified the scope is global. Whether scope is hierarchical or transitive along types, is not defined in the standard. Another association which states, that the *University of Bern* is also in *Bern* can be out of scope when considering only geographical facts.

Facet. The final feature of the topic map standard to be considered in this introduction is the concept of the facet. Facets basically provide a mechanism for assigning property-value pairs to information resources. A facet is simply a property; its values are called facet values. Facets are typically used for supplying the kind of meta data that might otherwise have been provided by SGML or XML attributes. This could include properties such as "language", "security", "applicability", "user profile", et cetera. Once such properties have been assigned, they can be used to create query filters producing restricted subsets of occurrences. This provides a complement to scopes; whereas scopes can be seen as a filtering mechanism that is based on properties of the topics, facets provide for filtering based on properties of the occurrences. Considering web sites about *Bern* with the assumption that the reader does not understand German. With the concept of facets, you can specify that only web sites in English are returned as a result of your search.

Topic Map Templates

All topics, occurrences, and associations can be seen as instances of classes (types). The classes themselves are expressed as topics. This class-instance relationship is in fact merely a syntactically privileged association type, as the standard makes clear: "The class-instance relationship ... could alternatively be established by a topic association link whose semantic is the relationship between a class and an instance of that class". This means that the class-instance relation is an association type predefined by the standard. Of course this is not enough. If we are looking at

the class-instance relation from an object oriented view, then there is a justifiable demand for a superclass-subclass relationship as well. However, the standard explicitly declares that such a relationship has to be user-defined.

When needed, maps can be merged. This allows users to use the concept of "templates" and logical modules in separate maps. A certain topic map A thus can include a *template map* B which consist of all constructs which have a declarative meaning for the map A - for example the basic topic types, occurrence roles, associations, and association roles from which topic map A only defines the instances. Other modules could consist of clusters of all typing topics for the various objects, the class hierarchy information, or consistency constraints. With the help of templates, the design and creation of topic maps can be split up into sub tasks. Furthermore, user access rights, user groups as well as roles can be assigned.

Relation Type	*Examples*
Superclass-subclass (indirect ancestor)	is a, equals, identical to
Superclass-subclass (direct ancestor)	is a, equals, identical to
Brothership	similar to
Kindship	less than, older than, closer to

Table 2.3: Relations in Class Hierarchies.

One practical application for templates are type hierarchies. We list some basic class hierarchy relationships in (Table 2.3). With such a type hierarchy, we could e.g., find implicit relationships along transitive associations like *is_a*.

More on templates and association taxonomies, class hierarchies or consistency checking can be found in a paper written by Rath [RATH 00].

Retrieve Information from Topic Maps

Posing a query in a way that the computer understands it quickly requires profound semantic background knowledge and in-depth expertise about the data model. An association taxonomy including meta information about relations like transitivity, symmetry, implication et cetera, can make queries look simpler, while they still are powerful. There is a standard for formulating topic map queries. It is called *Topic Map Query Language* (TMQL) in which queries resemble SQL statements ([KSIE 00]). Querying, as primary information retrieval discipline, is typically concerned with single uses of the system, by a person with a one-time goal.

Another discipline of information retrieval is filtering. Filtering is rather concerned with repeated uses by a person or persons with long-term goals or interests. The two concepts for filtering topic maps are scopes (domain, aspects like the user privileges) and facets (occurrence characteristics like the language).

2.1.3 Navigating Information

This section introduces navigation steps and issues of navigating information in knowledge management - again with the broader view on general information spaces, and not restricted to topic maps.

We classify navigation on different levels, according to the knowledge about the semantics in the model, in order to determine possible navigation steps and their applicability. First we consider navigation between tool states. Since topic maps are essentially graphs we then have a closer look at issues of navigation in graphs. Finally we stress the characteristics of navigation in models using specific information about the semantics, and show how navigation tracking can be used in a feedback loop to enrich the model.

Navigation between Tool States

The most general navigation is a sequence of user interactions and resulting tool states over time. Knowledge about the structure or semantics of the model is not needed for navigating from one tool state to the next, clicking on something, moving your mouse over something, or selecting a number of objects on the screen.

Among these navigation steps we can further distinguish between major and minor steps. The first category consists of steps that do modifications on the current model or selections, by changing the scope of the view, or the view itself. The second category consists of steps that only affect the visual representation of the same model, by moving the mouse over an object, zooming, or scrolling.

Considering sequences of actions and tool states we need a concept for modelling them. We want to memorize these sequences in a container and call that a *"session"*. Whatever a user does with his tool is stored in this session. Navigation steps between tool states - we can call them navigation steps on sessions - include Back, Forward, Peek, Undo, and Redo.

Navigation in Graphs

Navigation in graphs is basically navigating from node to node along edges. To generalize and qualify this neighborhood we introduce a new concept: the concept of proximity.

> **Definition:** ***Proximity*** between two nodes is the inverse of the number of associations that has to be traversed on the minimal path from one node to another. If there is more than one minimal path between two nodes, the proximities are accumulated. In the case that two nodes are totally separated we define their proximity to be 0.

Proximity is dependent of the mapping between domain model and graph. This simplified proximity is a hard fact. It is a measure that can be computed on every graph. More complex definitions of proximity would also consider further possible (not minimal) paths [PINT 95], though the computation of such a proximity can be hard.

Navigation based on Semantics

After having considered the way of navigation in an information space, independently from what the topics are, we now have a look at what the additional semantic information could contribute to enhance this most specific form of navigation, along real associations between topics. Since some of the associations are more relevant than others, we need a concept for rating associations. This leads us to the concept of affinity.

> **Definition:** ***Affinity*** between two topics is the weighted cumulated proximity of the corresponding nodes. To calculate affinity between two topics you must specify the weight of their association. A method can have a big affinity to one of its extenders or overriders.

Affinity is a soft measure. Semantics and experience is needed to find parameters in a form that the calculated affinity really conforms to the ideas of a user. Finding good parameters is difficult [PINT 95]. In different situations different configurations may suit better.

Good navigation support lets you quickly find the way from a topic A to other topics, having a big affinity to topic A. *Quickly* can be measured by quantification methods, developed in the field of Human Computer Interaction (HCI) [RASK 00]. Depending on the number of currently selected topics, more complicated associations are possible. Considering sets of selected topics, not only the number but also their type or the combination of their types is relevant. The large number of variations quickly reaches great complexity.

Navigation Tracking

> *"If you want to know where to lay a path between a new office building and its car park, cover the whole area with wood chips. Paths appear in the chips as each individual solves their own problem, and others can choose whether to use this solution. Within a short time a collective solution–a few well-used paths–emerges."*
>
> -Norman Johnson[6]

[6] Norman Johnson is the leader of the Symbiotic Intelligence Project at Los

A reasonable way to get parameters for calculating affinity between topics is to measure the popularity of navigation trails observed in previous investigations. Of course there is a danger of falsifications caused by the fact that also impasses or paths leading to red herrings attract future users. The usefulness of this feedback loop is dependent on the expertise and discipline of the early users of the system, if they are just clicking around, the results of navigation tracking will be of no use. It should always be clear which task the current user wants to perform. While searching dead code it makes no sense using trails gathered in sessions with the aim to detect duplicated code. Ideal navigation tracking records as much as possible, from mouse movements and clicks to currently selected objects, from performed actions to resulting tools states.

Another use of navigation tracking lies in gaining knowledge about heavily explored, or so far not visited areas of the system. We can draw navigation paths in a new kind of view. Storing the information about what entity is visited when, directly in the model, allows us to use *"intensity of observation"* as a metric. We can apply filters to hide them when exceeding a certain limit, color them accordingly, link them to other node characteristics, or take entities with a minimum metric value as a reduced subset for deeper inspections.

2.2 The FAMIX Reverse Engineering Model

After having discussed navigation in knowledge management, we now focus on issues in reverse engineering. Since this work is based on considerations made with the *Moose* reengineering environment (Appendix A), we present the model that *Moose* is based on. It is the FAMIX model [DEME 01] [TICH 01] on which a short overview follows now.

Various other formats to represent and exchange models of object-oriented code exist. The Rigi Standard Format (RSF) [WONG 98], is one of the more traditional ones. For a better interoperability between reverse engineering tools and components, various reverse engineering institutions formed a consortium to standardize a common Graph Exchange Language (GXL) [WINT 01]. GXL is an XML sublanguage.

The class hierarchy of all objects in a FAMIX model is illustrated in Figure 2.1. The core elements are entities and associations. The entities represent source code artifacts. The abstractions Class, Attribute and Operation from the Unified Modeling Language (UML), find themselves in FAMIX too. They are called accordingly Class, Attribute, and Method. The associations reflect relations between entities,

Alamos National Laboratory in New Mexico. The quote stems from an interview in Washington Technology. "A New Paradigm for Organizing Businesses". http://ishi.lanl.gov/Documents/interview.htm

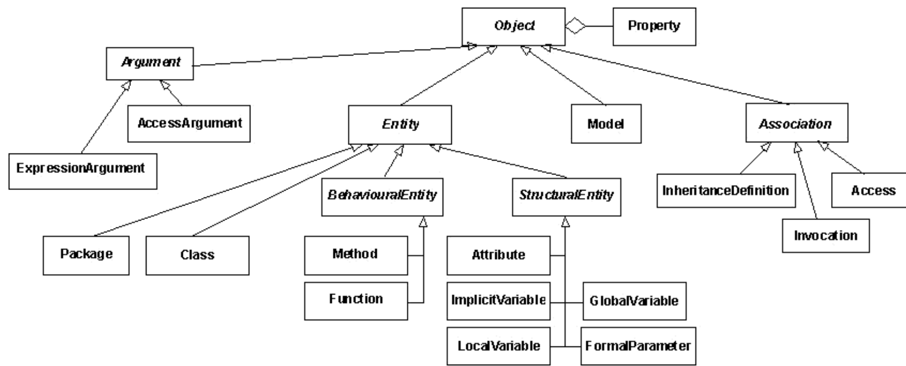


Figure 2.1: The Elements of the FAMIX Model.

which includes structural information about affiliations and about the class hierarchy, accesses and invocations. How the object-oriented entities and connecting associations work together, is presented in Figure 2.2. For mapping FAMIX models to topic maps we used topic map templates. One topic map consisted only of the structural definition of FAMIX entities, another topic map contained the relation taxonomy (including facts like the transitivity of inheritance), and a last topic map was filled up with the concrete instances of the FAMIX model.

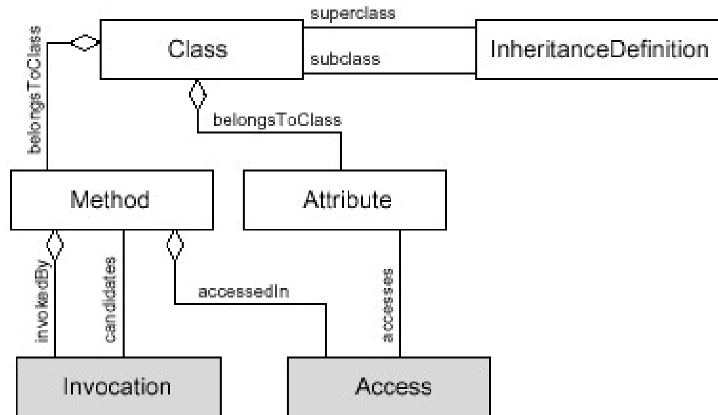


Figure 2.2: The Entities in the FAMIX model.

We briefly sketch how FAMIX models are mapped to graphs. Current graphical visualizations of FAMIX models [LANZ 99] map the objects of the model to nodes and edges as summarized in Table 2.4.

Having a closer look at the entities, the source code artifacts in FAMIX (Figure 2.2), we see that all structural information is represented through direct relations between structural and behavioral entities (belongsToClass). All other re-

FAMIX *Element*	*Node*	*Edge*
Class	×	
Structural Information **(including InheritanceDefinition and belongsToClass)**		×
Attribute	×	
Access		×
Method	×	
Invocation		×

Table 2.4: Mapping between FAMIX Elements and its Graphical Representations.

lations are represented by indirection through their own association entities (InheritanceDefinition). Note that this is not a must, but one possible way to do it. This decision has an impact on the concept of proximity, and the quantification of affinity. Relations could alternatively be modelled consistently, according to a one-to-one mapping - of the class hierarchy of the implementation - to nodes and edges. In the concept of topic maps you need not to decide whether a relation becomes a topic or just an association between two or more topics, since an association is just a specialization of a topic.

An example of good navigation support is when your are able to quickly find the way from two methods to attributes, which are commonly accessed by both, these two methods. We discuss the strategies for finding reasonable navigation steps and good parameters for affinity in Chapter 5.

2.3 Usability

"As far as the customer is concerned, the interface is the product."

-Jef Raskin [RASK 00]

After having described the *"what"* we now want to have a closer look at the *"how"* - this leads us to issues of usability, or more general to the field of Human Computer Interaction (HCI), and the underlying principles of psychology [WARE 00].

This section gives an overview of usability, although it is an extract and there is by far no claim for completeness. Only aspects that seemed interesting in the context of this work were regarded. We outline characteristics for the ideal and humane interface, show how efficiency can be quantified, and end up with a set of relevant factors for measuring usability.

The ideal human interface reduces the interface components of a user's work to begin habituation. Many problems that make products difficult and unpleasant

to use are caused by human-machine design that fails to regard the helpful, but also dangerous, properties of forming habit. Easy to use applications provide no multiple ways to accomplish one task, they support "Undo's" for any action instead of boring confirmation dialogs. They avoid confusing modes and cryptic shortcuts, but form automatizations and habit instead [RASK 00].

A superior interface is an exceptional long-term investment. IT returns not only higher productivity for customers, but also increased user satisfaction, a greater perceived value, a lowered cost of customer support, faster and simpler implementation, a competitive marketing advantage, simpler manuals or online help, and finally also safer products. In conformance with Raskin [RASK 00], we consider the following factors as measures of usability:

- **Simplicity** or **Ease of Learning.** We can measure usability by comparing the time it takes users to learn to do a job when working with an unfamiliar computer system to the time it takes them to learn to do the same job some other way. As measured by time, it takes the user more effort to learn a system that does not incorporate and build on the user's existing habits. The users will have to ignore what they already know about the job to develop a new collection of habits. Consistency, unification, standardization, and monotony help the users in creating habit, thus they simplify learning.

- **Efficiency** or **Ease of Use.** The minimum number of actions required to complete a task successfully becomes an increasingly important measure of usability for more experienced operators. For example, the number of mouse clicks entered per procedure is a good way to compare the ease of use of two designs. Other factors being equal, the design that requires fewer keystrokes per procedure will be more usable. There can be a trade-off between ease of learning and ease of use; consider the speed of execution of a shortcut-key-expert working with Emacs.

- **Complete Memory**. Any system must not harm your content at any time - user's input is sacred. A system must not lose or forget any input, action, or state. Memory must be physically storable and sharable.

- **Undo and Redo**. A system must not only remember any tool state in the past, but also provide ways to return to a previous point of interest at any time. Undoing and redoing actions are essential for an efficient system that does not harm your content.

- **Consistent Overview.** To hinder us from forgetting where we are, we need the support of a permanent overview. If possible, the overview must contain the whole system and show where our current focus lies, or which part of a view currently is visible in the current window. *"Geographical consistency"* is when an entity in a view is found at the "upper left corner" of

the whole system - it will always be there. Studies in the field of user cognetics [KITC 97] [RASK 00] prove that such cribs are helpful for humans to remember something and refind it. Arrangements in certain orders, e.g., alphabetical, can further simplify orientation.

- **Seamless Zooming.** To change quickly the viewpoint and switch granularity of detail, seamless zooming is needed. This is how we are used to work efficiently: looking at something in detail, going closer - then lying back and considering it in the context from a bigger distance.

- **Extensibility** and **Adaptability.** Aside from a good designed interface that regards the strengths and the weaknesses of humans - an interface that forms habit, there is another issue: Especially in areas of high complexity like reverse engineering where sessions vary by so many factors, tool designers cannot think of every feature that future users might need. There are also too many different programming languages, Integrated Development Environments (IDEs), or built-in constructs for source code management, that the model could be designed once - for all needs. In a small amount of time, users should be able to define their own views, navigation steps, menu entries, abbreviations, and do other adaptations.

2.4 Concerns Identified

With the common understanding about managing, modeling and navigating information and after visiting paradigms of usability, we are now ready to quickly summarize the most important considerations in three paragraphs. We end this chapter with a catalog of concerns that should be observed, when thinking about designing reverse engineering tools.

1. Navigation capabilities in object-oriented reverse engineering have their origin in decisions, taken at the time of modeling the information. Friction is induced because of several reasons, among them incompleteness of the model and features, indirection, red herrings, degradation of knowledge, and lack of relevance. We have discussed these in detail in Chapter 1.

2. Navigation can be performed at different levels. One classification of navigation steps according to the knowledge about the semantics of the underlying system, divides them into three groups: Navigation that is completely independent of the model, navigation on graphs from which we do not have to know what they are representing, and finally specific navigation in object-oriented systems. We cannot say which group of navigations is more important, all of them are necessary. The more we know about the semantics, the more complicated and individual the specific navigation features become. With increasing granularity they get more dependent on the reverse engineering task at hand and the characteristics of the current underlying subject system.

3. Another important issue in tool conception and design, is usability. Better usability, in general, is one of the major contributors for reducing friction in reverse engineering. Simplicity, efficiency, memory, rolling back, consistency, overviewing and zooming are key factors for usable tools. However, the task of reverse engineering is too complex, that you could think of all eventualities. An extensible architecture is helpful, for users can implement their additional needs with a minimum of effort.

We close this chapter with a catalog of concerns summarized in Table 2.5, whereas the individual concerns are itemized subsequently. This catalog will be revisited to identify strengths and weaknesses in our own reengineering environment *Moose* (Chapter 3) as well as for analyzing navigation and state-of-the-art implementations in Chapter 4.

	Concern
1.	**Low Entry Barriers**
2.	**Completeness**
3.	**Simplicity**
4.	**Navigation between Tool States**
5.	**Navigation in Graphs**
6.	**Navigation in Object-Oriented Models**
7.	**Efficiency**
8.	**Feedback**
9.	**Classification**
10.	**Complexity Reduction**
11.	**Consistency**
12.	**Memory**
13.	**Storage**
14.	**Extensibility**

Table 2.5: Concerns of Designing a Reverse Engineering Navigator.

1. **Low Entry Barriers.** Low political, psychological, and structural barriers for using the tool are essential. Simplicity, ease of use, a good user interface that is nice to use, and a clear predictable benefit from using the tool lower the barriers. Consistent undo-capabilities kill the fear of harming something or leaving unwanted traces.

2. **Completeness.** The completeness of navigation possibilities depends on the completeness of the model. Features for being able to access the whole amount of available information must be provided.

3. **Simplicity.** Ease of use, short time learning, nice-to-use, useful presentations of information, intuitive interfaces lower the barriers of using a tool.

4. **Navigation between Tool States.** Independently from the characteristics of the underlying model and the current content, a tool must provide possibilities to navigate between tools states. This includes going back, forward, return to the start, or undoing and redoing actions.

5. **Navigation in Graphs.** Graphs are a viable representation for information. They allow us to navigate from node to node along edges, using layout algorithms that arrange the nodes and edges in different ways. The concept of proximity within a graph is the base for the computation of affinity.

6. **Navigation in Object-Oriented Models.** This is the most specific kind of navigation. Using the semantics of object-oriented systems, new navigations can be identified. The concept of affinity can be applied to find related topics. For computing affinity an association taxonomy is necessary.

7. **Efficiency.** From the HCI point of view, efficiency is to perform a task within a minimum number of user interactions. With the help of affinity and statistics, reduced dynamic menus can provide abbreviations, metaphors and other

cribs for finding relevant information directly. For the most frequent sets of selections, we need a one-click-distance to the most likely next step. Undoing lets you quickly test things with the safe possibility to rollback - redo further increases your performance. Displaying automatically as much as possible of the most important detailed information, saves additional time.

8. **Feedback.** Unlearning (Table 2.1) is a basic driver for creating benefit in team working. It includes rating and annotating of topics and views. Automatic navigation tracking, and writing this information directly back to the model, is another technique that leads to something like round trip reverse engineering - where refactorings and restructurings, performed on the visual representation, find their way back into the code base, if desired.

9. **Classification.** A concept for the classification of topics and associations is important to know the value of direct and indirect navigations between topics. A classification can be obtained by rating topics and associations - it necessitates feedback, experience and knowledge about the specific task at hand.

10. **Complexity Reduction.** Reducing the amount of information reduces complexity. Filtering, zooming, aggregation, and diving in combination with relevance can help in concentrating on aspects, switching on and off the visibility of certain categories of information details.

11. **Consistency.** Geographical consistency is important for remembering and keeping a permanent overview. Consistency in menus, or generally, in the whole user interface, supports the creation of habit. Undoing should really result in the exact identical state from before, redoing should work as supposed. Information must always be up-do-date.

12. **Memory.** For not risking to harm any content, for relocating previous points of interest, as well as for navigation tracking, a memory is needed. This includes a history about all tool states, views, visited entities, user actions and movements or selections. Memory also traces feedback from users and lets you create your own bookmarks.

13. **Storage.** Memory must be physically storable, for sharing within a team, exporting to backup, and exchanging over a network or e-mail. This includes also verbose reports, the extraction of tables, or other data and statistics.

14. **Extensibility.** A tool must be adaptable to new situations: Domain-specific issues or tasks that cause changes in the model, new features, different affinity definitions, extended menus, refined rating mechanisms, new metrics, navigation paths, and personalized views - all this requires an architecture that makes extensions and adaptations easy.

Chapter 3

Context & Requirements

"Be your first customer!"

-This is our guideline for developing tools.

After having identified the concerns of building tools with the right navigation support for reverse engineering we now describe the environmental context and derive concrete requirements. We present *Moose*, our reengineering research platform (Section 3.1) and describe the experience we made in an industrial case study (Section 3.2). At the end of this chapter we put this described experience together with the theoretical concerns from the previous chapters. We compile a set of concrete tool requirements in Section 3.3.

3.1 Moose & CodeCrawler

Moose is our reengineering research platform based on the FAMIX meta model. Read more about *Moose* and its tools in Appendix A. FAMIX is a meta model that represents artifacts of source code in different programming languages with a special emphasis on object-oriented software.

Several tools were developed to provide different services to be performed on the *Moose* repository. Among the addressed fields are:

- Detection of duplicated code
- Analysis of runtime behavior
- Impact and dependency analysis
- Software visualization
- Software metrics
- Software evolution
- Refactoring and restructuring

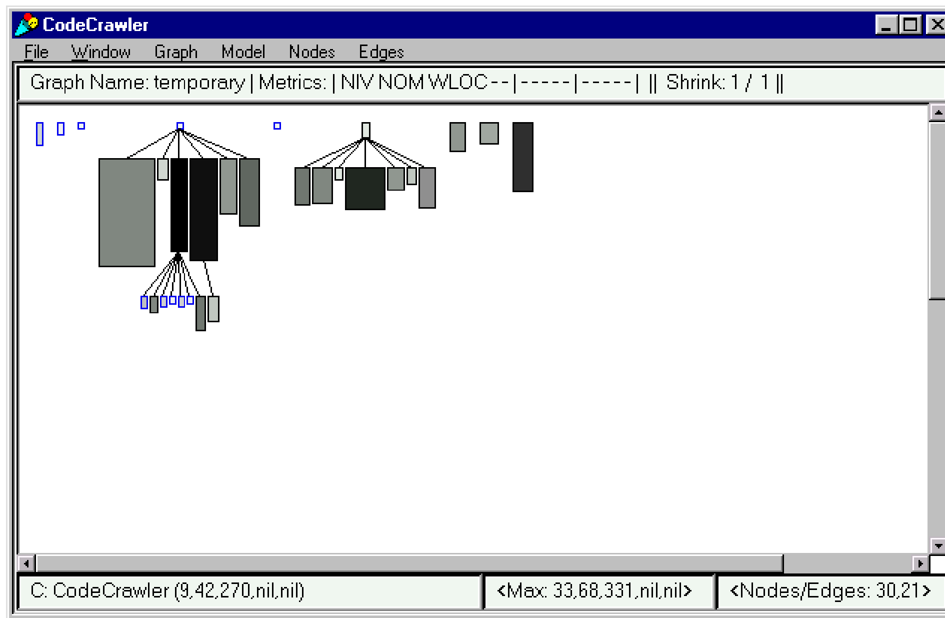


Figure 3.1: Screen Capture of *CodeCrawler*.

CodeCrawler is the visualization tool within *Moose*. It supports different views on a model, combining metrics and graphs [DEME 99] [LANZ 99]. The tool visualizes entities with shape and color according to metric values combined with different graph layouts. It enables a user to gain insights in large systems in a short time. Furthermore the graphs help a user to quickly identify source code entities with special combinations of metric values.

3.2 The SORTIE Experience

SORTIE is an established tool for modeling forest succession, implemented in C++. The system has evolved over a long period of time leading to a brittle architecture. Developers of state-of-the-art tools from research and industry analyzed the SORTIE system to recover the existing architecture, and to propose a new architecture. Read more about the project and see the full SCG report in Appendix B.

3.2.1 Experience

Loading SORTIE into *Moose* works fine. The system consists of 69 classes. From foregoing discussions we know that SORTIE is written with Borland C++ Builder, and none of the base classes is available, unless we buy a license of this product which we do not intend to do just for this experience.

For gaining further insights we use *CodeCrawler*. We start by creating a *System Complexity* view (Figure 3.2) to get an overview of the system. This view combines metrics and tree visualizations for locating hot spots [LANZ 99]. Hot spots help to find entry points and decide which parts to inspect in more details, or before the other parts.

Figure 3.2: SORTIE System Complexity.

In the *System Complexity* view shown in (Figure 3.2), each node represents a class in the system. The edges represent inheritance relations. Visual characteristics of the nodes are bound to metrics applied to the corresponding class: Node width $\simeq$ NOA (number of attributes of a class), Node height $\simeq$ NOM (number of methods of a class), Node color $\simeq$ supposed name spaces. This view shows a flat hierarchy. Most of the classes are standalone, and not clustered in class trees. After zooming the view to fit to the window size, the nodes are too small to work with. We miss an overview where we can see the whole system with a hint to the part that is currently visible.

We identify a number of "tall" classes. Among them we find the class *TMainWindow* which contains 237 attributes and 78 methods. It seems to be a kind of god class [BROW 98]. Diving into the class' internals in form of creating a *Class Blueprint* shows that there is one central method *RunSimulation()* which is basi-

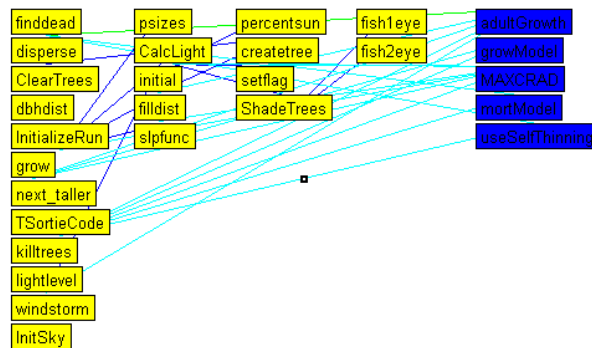


Figure 3.3: SORTIE Class Blueprint.

cally a procedural call of the complete program functionality. Another example *Class Blueprint* is shown in Figure 3.3, where nodes are either methods (yellow) or attributes (blue). From left to right the columns begin by listing methods according to their invocation depth - this is increased by one for each column. Thus we end up with: Interface methods, utility methods (multiple depths possible), pure accessors methods, attributes. Edges represent method invocations (dark blue) or attribute accesses (light blue).

The creation of views like the *Class Blueprint* that give insights into the internals of a class are complex and can be quite processing-time consuming. The ability for automation in creating the corresponding views for all the classes as a background process can help in faster navigation among the prepared views at a later moment of time. We write a small script generating all these views and are now able to parse a list of classes to quickly step through their *Classes' Blueprints*. Such an automation is ideally combined with the concept of a session which memorizes views, and supports managing views, and skipping from one to another. Such a session is also valuable to keep interesting views, like for guessing the system packages which we discussed above.

For quickly navigating we would like to have a concept of hyperlinks instead of user-action-costly procedure of selecting a node and then performing an action on that node via menu entries of the application window.

Even for us a help system describing the features of our tools is necessary, e.g., as a quick reference for the acronyms and names of metrics (like WLOC which stands for total number of lines of code). Another question that again and again raises is the naming scheme in the different types of views (like green which signalizes methods to be initializers).

To separate Graphical User Interface (GUI) and domain classes we create a

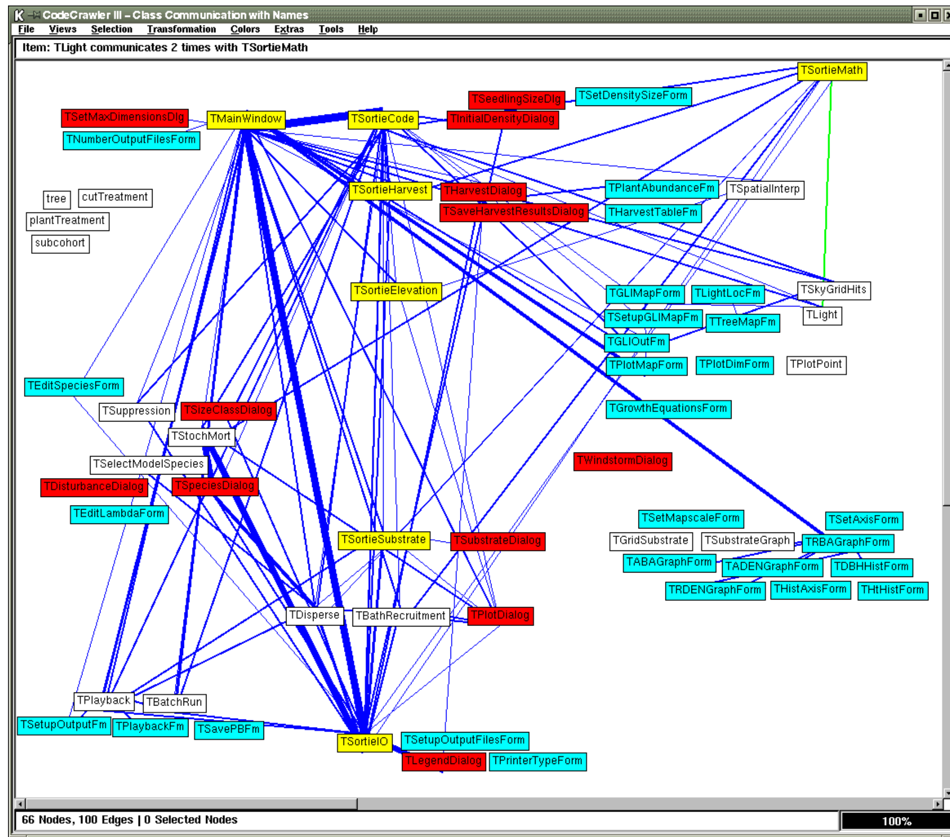


Figure 3.4: SORTIE System Packages.

Class Communication view. With this view we obtain an idea about which class is communicating how much with what other class. Applying layout algorithms that qualify edges, can help to arrange the classes according to their relation to each other. An example layout algorithm for that is the *Spring* [Gius 99]. We help the algorithm in manually arranging some classes into clusters. This is especially necessary for the standalone classes. Repeating these steps leads to an overview of probable system packages (Figure 3.4). In this view, each node represents a class. The edges represent communication between two classes. The thickness of the edges is in relation to the qualitative intensity (i.e., the sum of method invocations and attribute accesses). Colors were applied according to a speculation about the systems sub domains: GUI dialog classes in red, GUI form classes in blue, SORTIE core classes in yellow, structs and other not further specified classes in white.

A FAMIX model does not necessarily include the source code of methods. Depending on the level of detail while parsing, only the methods' signatures may be available. This turns out not to be enough for understanding what the system does.

After inspecting the system for a while we are guessing what parts of the system we have visited and what parts we might have overlooked so far. The current version of *CodeCrawler* does not support any functionality for tracking the navigation.

We work in a team of four people. Partly we work in pairs, partly each on his or her own. The current version of *Moose* does not support shared repositories and distributed working. So, each of us has to collect and extract the results by himself or herself. A phase of result integration and coordination is needed before the final report can be written.

Since the current version of *Moose* does not support the generation of reports, we collect the necessary data by inspecting the model, and reading the numbers and facts in our tool. Partly, we have to type the information manually in a text editor where we prepare the final report that has to be in HTML format. We use other tools to generate statistics and charts, and we use screen captures for illustrating interesting views, instead of having e.g., the ability to automatically generate pictures of all the views in the session.

3.2.2 Results

We summarize the main results of the SORTIE experience as follows:

- Factors for the completeness of the model are not always technical, but sometimes also of a political or financial nature.
- We are missing an overview of the system and the ability to know which part of the total system currently is visible.
- Metrics applied to views can save you the time to dive, or help you to decide which containers you want to explore in detail.
- Diving is necessary to understand a class' internals. A viable way of presenting the class internals is the *Class Blueprint*.
- We lack quick navigation based on a concept of hyperlinks.
- The ability for automation can increase the efficiency.
- We need a history of views between which navigation is possible.
- We need a help system - even for the experienced tool experts this is necessary.
- Layout algorithms like the *spring* can help you to reconstruct system packages, by arranging objects according to their interrelation. However, additional manual help is needed.

- You need the source code of a system in order to understand it - method signatures and invocations are not enough.
- We need a way to determine the coverage of visited and inspected parts of the system.
- We lack support for working in teams, like for instance a shared repository.
- We are missing better support for creating reports, and export sets of data.

Extensibility is a key feature for a tool that assists us in doing such complex and multifaceted tasks as reverse engineering a software system. The project specific needs are always different from what you expect, depending on the concrete task they want to perform. During our own case study we permanently extend and improve our environment for being able to cope with new requirements.

At this point we would like to underline the importance of working with tools that provide flexibility. E.g., tools written in Smalltalk do not need a scripting language, or API, and allow just-in-time adaptations. This has the advantage of shorter product lifecycles - short enough to be released before the case study is closed.

3.3 Tool Requirements

"You ain't gonna need it."
- *"Do the simplest thing that could possibly work."*

-Kent Beck [BECK 01]

Having a clear plan is a prerequisite to successful reverse engineering. However, it is difficult to know what you want to find out about a system. A proposition towards a methodology of exploring an unknown system [DUCA 01a] foresees four clusters of investigations: FIRST CONTACT, EXTRACT ARCHITECTURE, FOCUS ON HOT AREAS, and finally PREPARE REENGINEERING.

In search of ways to implement a reasonable support for navigation in object-oriented reverse engineering, we found that there is yet no methodology or cookbook for navigation support in reverse engineering tools. In this work we take a first step towards such a cookbook. We have collected problems and issues of navigation in object-oriented code in Chapter 2. This resulted in a list of concerns that an ideal reverse engineering tool has to address (Table 2.5). We analyzed and classified state-of-the-art solutions for navigation in Chapter 4. At the beginning of this chapter we summarized the main observations we made. Now we compile a set of tool requirements, by revisiting the set of concerns for a last time - providing

concrete answers and recommendations. For many issues we point to state-of-the-art solutions, and summarize how they can be implemented.

Many of the concerns depend on each other. The three concerns *Simplicity*, *Complexity Reduction* and *Extensibility* are especially linked to all others.

1. **Low Entry Barriers.** Low political, psychological, and structural barriers for using the tool are essential.

    - Clear benefit. A clear predictable benefit attracts users to work with a tool. This can happen when the tool really helps in better understanding a system. This can also be, because of the ability to generate views with the tool, illustrating the system's classes or components. The user should be supported in performing every task in one single tool, without having to juggle with data from one tool to the other. Finally a tool must allow easy, local and personal adaptations, so that users don't feel limited and patronized.
    - Installation Support. Installing a tool must be easy, without the need of making changes to the development environment or code base. The source code must be parsed without risking to harm it, or with the limit that while parsing current development must be paused. It is an advantage when the tool is integrated into the IDE, so that developers might incorporate reverse engineering in their daily work.
    - Short Learning Time. Short learning time is dependent on an habitual look & feel. Providing a demo for first-time users can lower time-to productiveness and the fear of not understanding how the tool works. Assistants & wizards help to perform a certain task for the first time, or complex tasks that are seldom used. Consistency in menus and actions is important to keep the overview and not being confused. A clear, integrated, and up-to-date help system is necessary.
    - Safety. Undo-capabilities mitigate the fear of harming any content.

2. **Completeness.** The completeness of a system is a prerequisite for users to build trust in a tool.

    - Model. On the one hand, the model must be complete, so that no available information is lost.
    - Features. On the other hand, the set of features must be adequate; for accessing all this information; for navigating from one piece to the next; for creating useful views, and finally for printing nice reports.

3. **Simplicity.** A good user interface is intuitive, shortens learning time, and is fun to use.

- Simple Manipulation. Simplicity helps in forming habit. Ease of use, simple navigation, intuitive menus and buttons, are sample requirements. Users must be able to easily perform their tasks, and rapidly form habit. Efficient navigation is provided with short interaction distances to related information (Section C.2.6).
- Simple Representation. Useful graphs and views help in reducing complexity and showing the system from new perspectives, e.g., at higher levels of abstraction (Section C.2.5).

4. **Navigation between Tool States.** A good tool provides ways for managing tool states and actions. It memorizes what we do, and where we go.

   - Creating Tool States. For being able to go back to a certain position the memory remembers every tool state. Features include: Back, Forward, First, Last, or selecting a specific tool state from a list.
   - Manipulating Tool States. By zooming, scrolling et cetera existing tool states can be manipulated marginally, without the need of creating a new one.
   - Actions. Another useful feature is when a tool lets us redo and undo actions. This is not trivial, since the reasoning about what exactly a user wants to undo or redo is difficult, and many actions are hard to discard or repeat.

5. **Navigation in Graphs.** If a tool represents information in the form of graphs, a series of features must be provided to enable navigating from node to node along edges.

   - Neighborhood. Finding neighbor topics in navigating along associations lead to the concept of proximity within a graph.
   - Hyperlinks. Selecting nodes and wandering in a new graph where the selected node is the one with focus is an intuitive way for navigation.
   - Layouts. Graph layout algorithms help in creating useful views. Different levels of detail or aspects call for different representations of graphs - while we find a tree useful for representing class hierarchies it is not the layout to display a methods call graph - this is better done with a *Class Blueprint*.

6. **Navigation in Object-Oriented Models.** Considering the specific semantics of object-oriented systems, a new set of navigations results.

   - Fixed Navigations. Finding neighbor topics in navigating along associations leads to the concept of proximity within a graph. Proximity is the base for computing affinity. Navigating along affinity.

- Affinity. Based on proximity within a graph, affinity can be an efficient criteria for finding useful navigation steps, though it can be difficult to determine felicitous affinity parameters. Usually an association taxonomy is inevitable.

7. **Efficiency.** From the HCI point of view, efficiency is to perform a task with a minimum number of user interactions.

    - Pushing Information. Displaying automatically as much as possible of the most important detailed information, safes additional time in making extra accessing superfluous. You can use *mouse-over* events to display details about underlying elements.
    - Pulling Information. Short ways to find information, are provided by abbreviations, cribs, or other metaphors - they must be configurable and extensible. They can be acquired with the help of proximity and affinity, but also via feedback loops - rating or navigation tracking.
    - Near Features. To avoid red herrings, reduced clear dynamic menus only provide the most frequent or likely next commands. Other, more seldom used commands, are possible, but hidden in deeper menu hierarchies. The most likely next step can be performed ideally by a minimum number of user interactions, e.g., by double clicking (Section 5.1). Undoing lets you quickly test things, with the safe possibility to rollback, this saves time. Also can redo further increase your performance.
    - Automation. The programmability per API or scripts can reduce the effort of performing recurring tasks. These "macros" can also perform computing-time-intensive tasks, e.g., during night.

8. **Feedback.** Unlearning - the process of *making-explicit-to-others* of what you learn (Table 2.1) - is a basic driver for creating benefit of team working. New metrics can open new fields of views about a system - showing new aspects. Metrics can be set in the form of ratings, but also automatically, e.g., metrics about navigation paths.

    - Manual Feedback. The rating of topics and views helps to increase the precision of affinity. Leaving annotations, considerations, questions and experiences directly in the model, leverages the total information about a system. Sometimes tasks or problems are too big or too complex, that one single person would be able to solve them alone. Dividing tasks, to be solved easily, and reintegrate the different sub tasks to one single result set, allows us to solve bigger and more complex problems than one single person could cope with. Also for quality assurance reasons one should not let one single person do crucial tasks. Annotations and ratings of colleagues can help in avoiding making the same errors or doing work twice.

- Automatic Feedback. Another way of feedback is the automatic enrichment of the model with statistics about user behavior. This lets us locate so called trails and detect so far untried areas of a system. In general, feedback loops help in improving the efficiency of navigating a system, based on the popularity and experiences of past investigations. It can help us to identify blind alleys that are taken often, or other problems with the tool.
- Round Trip Engineering. Ideally, a tool provides the possibility that modifications on the model or visual representation, find their way back into the code base, if desired. Round trip engineering without harming either design or code is difficult, though.

9. **Classification.** A concept for classifying topics and associations is important to determine their relevance, or the value of direct and indirect navigations between topics.

    - Explicit Classification. A classification of relevance can be obtained in rating topics and associations - it necessitates feedback, experience and knowledge about the specific task at hand.
    - Implicit Classification. Based on the popularity identified in tracking the user behavior in previous investigations, a system can try to automatically classify topics and associations. Knowledge management systems that seek automatically for affinity using the techniques of *Artificial Intelligence*, are yet in their beginnings.

10. **Complexity Reduction.** Reducing the amount of information reduces complexity of the information glut.

    - Meaningful Representation. Using good layout algorithms, aggregating, zooming, or diving, you can switch on and off the visibility of certain categories or levels of information details. By decorating, coloring, highlighting or applying metrics to forms and figures, you further achieve better and faster recognition of topic and association characteristics.
    - Reducing Volume. To get simpler views you can hide certain aspects or concerns by filtering. The variety of possibilities for doing that increases, by a foregoing process of rating, since you can use this meta information in a feedback loop, as an additional criteria to further reduce the volume.

11. **Consistency.** Consistency is important for building trust in an environment, and keeping the overview.

    - *Geographical Consistency* (Section 4.3). It can help to keep the overview of the whole system, and for finding back to previous points of interest.

- User Interface. A consistent user interface helps in forming habit, and thus, ease of use. This includes consistency in menus, in names, and in presenting information.
- Content and State. No content must be harmed at any time. Going back or undoing actions must lead to exactly the same content and state as before. This comprises also a possibility to freeze a tool state - to leave a system and refind it in exactly the same state in a future moment of time, or even exchange a complete model from one person or machine to the other, so your colleague can tie up working, exactly where you stopped.

12. **Memory.** For not risking to harm any content, and for better refinding previous points of interest, as well as a base for applying concepts for a classification of relevance, a memory is needed.

    - Recovery. The memory remembers every view, tool state, positions, actions, mouse movements, and visited entities, or selections. If consistency is fulfilled, you are able to easily find everything back. Instruments for that include bookmarks, sessions or histories, and the management of such.
    - Model Enrichment. Implicit and explicit classifications can be used to determine relevance within a system, according to a certain concern. Annotations or comments must be memorized in the model.
    - Team Support. A shared memory is needed to collaborate within a team. Ideally this allows you to work concurrently, while you see - in real time - the modifications, questions, and considerations of your colleagues.

13. **Storage.** Memory must be physically storable, for safety and collaboration reasons.

    - Safety. For being able to store models and insights for the future, the gathered information must be saved in a file that can be duplicated for copies, backup et cetera.
    - Exchangeability. We might want to exchange models and insights. A first step towards collaboration, is having the possibility to save the current work and later tie up where the work has been stopped. Another step, includes being able to work on a shared repository. The most sophisticated edition, lets you work in separated places, and merge the work when you are connected again.
    - Reports. Finally you want to create nice reports of your session. You want to print documents, including different statistical analysis, and nice views, without the need to take screen captures. Lists of visited entities, metrics applied to certain subsets, et cetera, should be ready

to be exported in a common file format, for the case that you want to process them in other tools. Navigation logs, annotations and the like must also be available for reporting.

14. **Extensibility.** The systems and tasks in reverse engineering are too diverse, that one single implementation can be found to meet optimally all requirements. A tool must be open and easily extensible, to make all users satisfied.

    - Model. Domain- and system specific issues may require an extension or adaption of the model.
    - Tool. New features, menus, navigations can be useful to increase efficiency or to enable performing new tasks.
    - Concerns. Under different concerns, different parameters count for determining relevance or affinity. New metrics might be needed to cover the new aspects. Personalized views can better fit to the current task.
    - Export. What you want to export out of your reverse engineering session is personal and can vary. The format of information must be configurable (reports, tables, lists, et cetera)

This is the end of the set of requirements. In the next section we present some example implementations on the base of our prototype *MooseNavigator*.

Chapter 4

Classifying Navigation

> *"The real danger is not that computers will begin to think like men, but that men will begin to think like computers."*
>
> -Sydney J. Harris (1917-1986), journalist & author

The previous chapters were about introducing the general concepts of navigation and explaining the context. We compiled a set of requirements for measuring state-of-the-art navigation tools. In this chapter we split navigation into its building blocks and introduce a terminology for classifying concrete solutions and features of existing reverse engineering tools. The chapter ends with a summary of Appendix C which is the evaluation of a set of state-of-the-art navigation and reverse engineering tools.

We begin by introducing two utilized concepts - the concept of *views* and the concept of *navigation steps*.

> **Definition: A *view*** is a visual representation of a subset of a model. In our case these are usually graphs on which a certain layout is applied. A typical view is a class hierarchy tree.

> **Definition: *Navigation steps*** are the atomic building blocks of navigation. We distinguish *Navigation Steps between Tool States* and *Semantic Navigation Steps*. A typical navigation step between tool states is pushing the *Back*-button in a web browser. A typical semantic navigation step in an object-oriented model is to navigate from a class to its superclass.

In the following two sections we list navigation steps of each group. The presented lists contain steps that we have seen by example in the state-of-the-art in Appendix C; navigation steps observed in other tools; navigation steps that we have implemented in our own tool (which we will describe in Chapter 5); and finally candidate navigation steps, from which we know - through our daily reverse

Navigation Step	*Description*
Session First	Return to first tool state of the current session
Session Back	Return to the previous session state. Especially interesting is to observe which actions preceded going back - these actions are candidate blind alleys and red herrings, attracting the user but turning out to bring no value
Session Peek	Return to certain session state in the history list
Session Forward	Jump to the next session state
Session Last	Jump to the next session state
Change Focus	Change the focus in the current view, my moving the mouse cursor over a figure
Change Selection	Change the selection in the current view, by clicking on a figure or selecting a number of figures alternatively
Change Extract	Change the visible extract of the current view, by scrolling or zooming
Change Layout	Change the layout of the current view, by rearranging the figures applying another layout algorithm, collapsing or out folding, coloring nodes of a certain kind, transform nodes or edges, grouping them, applying new metrics to their characteristics et cetera
Change Filter	Change the currently applied filters on the view, and hide or unhide figures of certain kinds
Dive	Reduce model to selection, throw away the rest
Crawl / Change View	Create a new view on the same model
Pop	Return from a subset of the model back to the complete one
Spawn	Create a new window, clone and display the current view in it
Undo	If the previous user interaction was a minor step there is nothing to undo, if it was a major step then *Undo* is equal to *Session Back* with following elimination of the latest view.
Redo	If the previous user interaction was a minor step there is nothing to redo, if it was a major step then try to abstract the previous user interaction and repeat it.

Table 4.1: Navigation between Tool States.

engineering experience and experiments (Chapter 6) - that they can be of use. At the end of this chapter we present a summary of the evaluated state-of-the-art tools and show in what respect they address the compiled concerns of navigation.

4.1 Navigation Steps between Tool States

Considering the process of reverse engineering, we usually need quite a lot of time to understand one single view, and identify all relevant information. Usually we need also quite a lot of time to come to a decision about what we want to see next, and how this should be presented. In this work we do not further focus on issues of computer supported decision support.

People tend to have problems with remembering hard and exact facts like numbers, positions, series et cetera, especially for many things simultaneously

[Rask 00]. The bottleneck of progress in understanding is human thinking, not the performance of the tool in presenting new views. On the other hand this is exactly what computers are good for.

> **Definition: A *session*** is a list that holds a series of tool states over time in form of views, and the actions that lead to the specific tool states in form of navigation steps.

When we once have created some interesting views, we want to be able to switch quickly between them, to compare them, to combine them, zoom in, zoom out again, or test something with the aim of undoing it immediately afterwards. We collected all these possible navigation steps between tool states in Table 4.1

4.2 Semantic Navigation Steps

In this section we list semantic navigation steps in models of object-oriented code. For compiling them we also inspect ways of automatically extracting possible navigation steps from a given situation. Since we focus on navigation in object-oriented reverse engineering, semantic navigation is navigation along interdependencies of object-oriented code artifacts. In an experiment we tried to scan the meta model for the artifacts' relationships. The experiment resulted in the perception that this still needs a lot of human input. Read more about the experiment in Section 6.1.

We present navigation steps that have their origin in semantic relations between artifacts in the model. The following lists of navigation steps are grouped according the type of the particular artifact where a navigation step starts, the discussed types include *class*, *attribute*, and *method*, since these are the basic building blocks of every object-oriented system.

If the current focus is on an entity representing a **class** in an object-oriented system, some of the most popular navigation steps are: *Attributes, Methods, Fullclass, Superclass, and Subclass*. See a extended list of frequently used navigation steps in Table 4.2 . In dynamically typed languages like Smalltalk, an extra effort is needed to find *Deleted Classes*, *Overhanded Classes* and *Returned Classes* for navigation steps starting on entities representing classes or methods.

Starting on an entity representing an **attribute**, reasonable next navigation steps include: *Class, Accessors, and Initializers*. Table 4.3 lists further popular navigation steps for this situation. In dynamically typed languages an extra effort is needed to find *Value Classes* as a navigation step starting on attribute-entities.

Having the current focus on a **method**-entity, we might want to find out its *Class, Senders, Implementors, Overriders, or Extenders*. We list other possible navigation steps starting on method-nodes in Table 4.4.

Navigation Step	*Description*
Attributes	Attributes of a class
Inherited Attributes	Attributes that a class inherits from one of its superclasses
Additional Attributes	Attributes of a class' class or metaclass
All Attributes	All attributes of a class, including all methods of its superclasses
Accessed Attributes	Attributes of other classes that are directly accessed by a class, this is not possible in many programming languages
Methods	Methods of a class
Inherited Methods	Methods that a class inherits from one of its superclasses
Extender Methods	Methods that extend superclasses' methods
Overrider Methods	Methods that override superclasses' methods
Overridden Methods	Methods that get overriden by subclasses
Accessor Methods	Methods of other classes that are directly accessing a class, this is not possible in many programming languages
Invoker Methods	Methods of other classes that are invoking methods of a class
Invoked Methods	Methods of other classes that are invoked by methods of a class
Additional Methods	Methods of a class' class or metaclass
All Methods	All methods of a class, including all methods of its superclasses
Fullclass	Fullclass of a class, which is all attributes and all methods not only from the class itself, but also all inherited attributes and methods from the classes superclasses, except the attributes from the object root (e.g. the class *Object* in Smalltalk)
Direct Subclasses	Direct subclasses of a class
Subclasses	Subclasses of a class
Direct Superclass	Direct superclass of a class
Superclasses	Superclasses of a class
Accessing Classes	Classes with Accessing Methods
Accessed Classes	Classes with Accessed Methods
Invoking Classes	Classes with Invoking Methods
Invoked Classes	Classes with Invoked Methods
Created Classes	Classes of which instances are created in a class
Deleted Classes	Classes of which instances are deleted in a class
Overhanded Classes	Classes of which instances are overhanded as a parameter value of a method in a class
Returned Classes	Classes of which instances are returned from methods in a class
Class Tests	Unit tests covering a class (Granularity can be Methods, Classes, Applications etc)
Related Classes	Subclasses, Superclasses, Accessing Classes, Accessed Classes, Invoking Classes, Invoked Classes, Created Classes, Deleted Classes, Overhanded Classes, Returned Classes
Related Stuff	Owner, SRS, other Documents, Source File, Configuration Map, etc

Table 4.2: Navigation Steps Starting from a Class.

Navigation Step	*Description*
Class	Class defining an attribute
Accessing Methods	Methods accessing an attribute
Accessing Classes	Classes defining Accessing Methods
Accessor Methods	Methods uniquely for accessing an attribute (Setters & Getters)
Accessor Classes	Classes defining Accessor Methods
Initializer Methods	Methods initializing an attribute
Initializer Classes	Classes defining Initializer Methods
Values Classes	Classes of objects that are stored in an attribute

Table 4.3: Navigation Steps Starting from an Attribute.

Navigation Step	*Description*
Class	Class defining a method
Senders	Classes invoking a method
All Senders	Classes invoking a method, including the subclasses that can send methods via inheritance
Implementors	Classes implementing a method
All Implementors	Classes implementing a method, including the subclasses that know how to process a method by delegating it recursively to its superclass
Super Method	Method that is overridden by a method
Super Method Classes	Classes defining the Super Methods
Overrider Methods	Methods of subclasses that override a method
Overrider Classes	Classes defining Overrider Methods
Extender Methods	Methods of subclasses that extend a method
Extender Classes	Classes defining Extender Methods
Accessed Attributes	Attributes accessed by a method (Granularity: Local, All)
Accessed Classes	Classes defining Accessed Attributes
Invoked Methods	Methods invoked by a method (Granularity: Local, All)
Invoked Classes	Classes defining Invoked Methods
Created Classes	Classes of which instances are created in a method
Deleted Classes	Classes of which instances are deleted in a method
Overhanded Classes	Classes of which instances are overhanded as a parameter value of an invoked method in a method
Returned Class	Class of which an instance is returned from a method
Method Tests	Unit tests covering a method (Granularity can be Methods, Classes, Applications etc)

Table 4.4: Navigation Steps Starting from a Method.

4.3 The Dimensions of Navigation

In this last section of classifying navigation we introduce the concept of dimensions in navigation. We exemplify the given definitions by navigation steps which were introduced earlier in this chapter.

> **Definition:** ***Horizontal navigation*** is changing the focus from one topic to another topic, without changing the level of detail. A *horizontal* navigation step is *Direct Superclass* performed on a hierarchy tree of a class.

> **Definition:** ***Vertical navigation*** is changing the level of detail without changing the topic under focus. A *vertical* navigation step is *Methods* performed on a hierarchy tree of a class, leading to a representation of this single class' internals.

> **Definition:** ***Diagonal navigation*** is navigating *horizontally* and *vertically* simultaneously. A *diagonal* navigation step is *Overridden Methods* performed on a hierarchy tree of a class, leading to a representation of the internals of all the classes that implement methods that override methods of the original class.

Another issue in navigation and keeping the overview is a consistent way of presenting data. There are tools that display objects always in the same manner in respect to their relative positions. In such tools users easier relocate objects, since they can better remember their positions with the help of cribs or other metaphors like *"the slim, dark object in the upper left corner, on the left of the tall one"* [KITC 97] [RASK 00].

> **Definition:** ***Geographical consistency*** is the characteristic of a tool to display objects always in the same manner in respect to their relative positions.

4.4 State-of-the-Art Navigation

To study best practices in navigation we evaluated several state-of-the-art tools. We discuss nine of them in detail in Appendix C. This section is a short summary with the exclusive focus on supported navigation features.

All the evaluated tools support the modification and interaction with the generated views. Most of the tools provide navigation within a history of tool states including *Back* or *Forward*, whereas a history of actions and the possibility of *Undoing* or *Redoing* actions is not equally widespread.

Except from Eclipse and Javadoc all tools pursue a concept of neighborhood and represent artifacts and relationships visually as graphs of nodes and edges. Graph layout algorithms help to arrange the nodes and edges in different ways.

Hyperlinks are a popular concept for navigation. Efficient navigation is when the system can propose short-cuts for these often used series of commands and provide navigation steps for directly getting there. For this semantic information about the model is needed. Some tools use a concept of affinity which can be used to find related information or as an input for layout algorithms to group related entities and identify clusters.

Many tools do not provide *geographical consistency* or a permanent overview on the complete system.

Table 4.5 gives an overview of the evaluated tools and shows which features are supported. This table is an extract of Table C.7. The paradigm in the first columns correspond to the tool requirements for the three concerns *Navigation between Tool States*, *Navigation in Graphs* and *Navigation in Object-Oriented Models* which were compiled in Section 3.3. Read more about the tools in Appendix C.

Legend for Table C.7

⊖	Not supported / missing - Insufficient
∼	Supported with reservations - Sufficient
√	Satisfactorily supported, implemented, available - Good
N/A	Not Applicable or Unknown

Paradigm	Eclipse	Javadoc	Rigi	SHriMP	Small Worlds	TheBrain	Together
Tool State History	√	∼	√	√	√	√	∼
Manipulate Tool States	√	√	√	√	√	√	√
Actions History	√	⊖	⊖	⊖	⊖	√	√
Neighborhood	⊖	⊖	√	√	√	√	√
Hyperlinks	∼	√	√	√	∼	√	√
Graph Layouts	⊖	⊖	√	√	∼	√	∼
Semantic Navigation	√	√	√	√	√	√	√
Affinity	⊖	∼	√	√	√	√	∼
Geographical Consistency	N/A	N/A	√	√	√	√	√
Overview	N/A	N/A	√	√	√	⊖	√

Table 4.5: State-of-the-Art Navigation Overview.

Chapter 5

Navigation for Reverse Engineering Tools

> *"It is impossible to make anything foolproof because fools are so ingenious."*
>
> -Edward A. Murphy, engineer

This document focuses on navigation in object-oriented reverse engineering. The previous chapters build the foundations for our work. We encompassed the broader context of reverse engineering and navigation, discussed the theory of managing, modelling, and navigating information - and introduced some relevant issues of human computer interaction. This resulted in a set of concerns that an ideal reverse engineering navigator has to address, listed in Chapter 2. In Chapter 3 we described our experience with our own reengineering environment during a case study. We identified strengths and weaknesses concerning navigation and compiled a set of requirements. After that we analyzed and classified issues and best practices of state-of-the-art tools in navigation (Chapter 4).

This chapter brings all this information together, we first summarize the main perceptions made in this work. After that we present *MooseNavigator*, and show prototypically for some of the previously compiled tool requirements how they could be realized in our environment.

5.1 Perceptions

During this work we learned a lot about navigation in reverse engineering. We evaluated many tools, and we set up several experiments. In this section we describe the most important perceptions on the users' behavior in reverse engineering and adequately efficient navigation support.

The complete set of possible navigation steps on a model of object-oriented code is huge. The distinction between useful and useless navigation can only be made with the help of knowledge about the semantics of the model and human experience in the domain. We did a small experiment on that issue, it is described in Section 6.1.

Efficiency is to perform a task within a minimum number of user interactions (Section 2.3). Consequently, for building an efficient navigator, we must know the tasks, and provide short-cuts for these often used series of commands. We tracked how users navigate a system in reverse engineering. The experiments are described in Section 6.2 and Section 6.3.

Here are the main observation we made, we describe them in more detail subsequently.

1. Users often visit detailed information.
2. Users complain about inconsistent or incomplete functionality.
3. Users navigate slowly.
4. Users usually navigate only in one dimension at a time.
5. Many features are seldom used.

In giving the users the possibility to permanently displaying detailed information about the current entity we could reduce the necessary number of navigation steps, we even achieve a "zero-interaction-distance" for the task of inspecting normal detailed information.

When users detect missing functionality their focus of attention changes from the task of reverse engineering a system to the task of evaluating the reverse engineering tool. This user distraction decreases efficiency and increases psychological barriers to work with the tool.

Users do not quickly create new views, but rather spend a lot of time to understand a current view. When they navigate quickly, it is between previous tools states, for comparing views that they already know to a certain extent. Users prefer *vertical* and *horizontal* to *diagonal* navigation which includes changing multiple dimensions simultaneously is seldom performed. Changing one dimension at the time, is apparently complex enough. Users that have to cope with more information than they can understand at once, try to catch the complexity and the interrelations by scrolling, zooming in and out, by hiding and un-hiding details, diving and popping. Raskin calls his answer to this phenomenon *Zoomable Interface Paradigm* (ZIP) [Rask 00]. The ZIP suggests to present the whole available information on a two dimensional pane. On that you navigate with a magnifier, of which you can

adjust the strength. We adopted a similar approach in *MooseNavigator*.

The most frequently used tasks we would like to reach by a "one-interaction-distance". Since users tend to use only a few views, and these are always the same, we want to provide short paths between them. Concretely this means that we want to provide transitions from one view to another by double clicking figures. We identified the following navigations to cover a reasonable part of the needs:

- System Overview. Normally users start with an overview of the system to see class hierarchies and detect eye-catching classes. An example of this type of view is *System Complexity* [LANZ 99].
  - Class node. Double clicking a class node in this view results in a presentation of a *Class Blueprint* view of the according class.
  - Inheritance definition edge. Double clicking an edge representing an inheritance definition in this view results in a presentation of a *Class Blueprint* view containing the according class and all its superclasses.
- Class Internals. This is the view that presents the internals of a class. Here the user can see the attributes and the role and collaboration of the methods. The view can also be applied to two classes, showing the overall cooperation. An example of this type is the *Class Blueprint* [LANZ 01a].
  - Class node. Double clicking a class node in this view results in a presentation of a *System Complexity* view of the class' root and all its subclasses.
  - Method node. Double clicking a method node in this view results in a presentation of another *Class Blueprint* view, showing all involved classes, which are classes that have a direct relationship - in form of accessors or invocations - to the according method.
  - Attribute node. Double clicking an attribute node in this view results in a presentation of another *Class Blueprint* view, showing all involved classes, which are classes that have a direct relationship - in form of accessors - to the according attribute. In many programming languages the access of an attribute from outside the class is not possible, in this case this results in another *Class Blueprint* view of the single class.
  - Edge within a class. Double clicking an edge within a class in this view results in a presentation of a *System Complexity* view of the class' root and all its subclasses.
  - Edge connecting two classes. Double clicking an edge that connects two class in this view results in another *Class Blueprint* view, showing these two involved classes.

All other navigation features can still be performed by using the conventional menu entries. To formulate more complex queries, especially for creating specific data sets we suggest rather using query tools like *MooseFinder* and *MooseExplorer*, and running the predefined queries, fetching the result sets and proceed with them afterwards again in navigation tools like *MooseNavigator*. If this is not enough, or the tasks at hand call for other navigation short-cuts, the tool must be extensible and adaptable, so that this can be changed within a short amount of time. This leads us to the next issue:

Extensibility is a key feature for a tool that assists us in doing such complex and multifaceted tasks as reverse engineering a software system. The specific needs of users may vary, depending on the concrete task they want to perform [FAVR 01]. The ability of permanently extending and improving the environment to cope with new requirements can be a crucial success factor for a case study. We describe our own experience that validates this statement in Section 3.2.

5.2 A Prototype - MooseNavigator

For better being able to study issues of navigation in object-oriented reverse engineering we needed a tool for testing ideas, implementing features, and to observe and record the behavior of users. This was the motivation to build *MooseNavigator*, a prototype reverse engineering navigator. *MooseNavigator* is an extension to *CodeCrawler* which combines metrics and graphs to visualize software systems [LANZ 99]. *CodeCrawler* is one of the tools of the *Moose* reengineering environment [DUCA 00a] [DUCA 01b] [TICH 01]. More information about *Moose* and *CodeCrawler* can be found in Section 3.1, Appendix A, or online[1].

For every component of *MooseNavigator* we briefly indicate which concern (Table 2.5) it addresses or what tool requirement (Section 3.3) it implements.

5.2.1 Conception

While *CodeCrawler* was designed under a lightweight approach with the primary aims of scalability and simplicity, the constraints for *MooseNavigator* were different. Scalability or performance were not in the foreground, but support for navigation, orientation, and efficiency. *MooseNavigator* offers purposely a wider range of partly redundant features for studying which features are preferred by users. *MooseNavigator* concurrently displays a lot of detailed information in one single application, with the aim to reduce complexity by supporting interactive and dynamic navigation in a subject system. One of the main differences to *CodeCrawler* is that an application is not tied to one single view, but holds a collection of views. This collection is stored in a user session. A system can be browsed like surfing

[1] http://www.iam.unibe.ch/~scg/

along the Internet, clicking on entities and diving into its details, or jumping backwards and forwards within the list of previous views.

The three central concerns *Simplicity*, *Complexity Reduction* and *Extensibility* guided us through all the design decisions. We implemented many navigation features which can not be assigned to one particular component of the system, such as *Neighborhood* or *Hyperlinks*. We start by presenting the user interface followed by some other key features and components.

5.2.2 User Interface

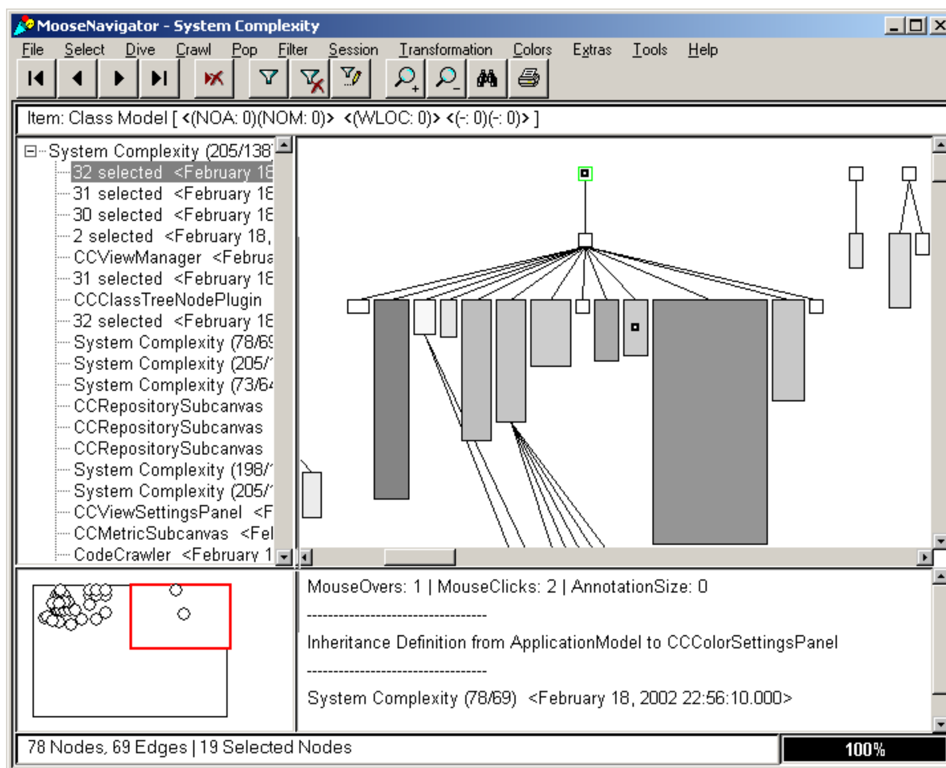


Figure 5.1: Screen Capture of *MooseNavigator*, Tools In-lined.

Figure 5.1 shows the main window of *MooseNavigator* with all its sub tools integrated in a single window interface. The proportions of each sub tool can be varied by two sliders: a horizontal and a vertical one. In-lining the tools is an option, all of them can be run as separate windows alternatively. In the following we describe in detail the separate tools.

The primary concerns which guided us through the design of the user interface were *Low Entry Barriers*, *Simplicity*, *Efficiency*, and *Consistency*.

Main Window

Like in *CodeCrawler*, the main sub window of *MooseNavigator* is the drawing, which displays the current view (Upper right pane in Figure 5.1). The toolbar contains buttons for accessing navigation and other features directly (from left to right): Session first, session previous, session next, session last, session remove current, switch current filters on, switch current filters off, open filter library editor, zoom in, zoom out, find entities by block, print current view. Clicking on the view pane, or on a node- or edge-figure is considered a major change of the current point of view, and thus causes a new session state to be created. A double click on a node figure, automatically causes the system to perform the next, most likely navigation step. This is, for example, to display the class' blueprint when the double click is performed on a node in the system complexity view. The context menu which pops up on right mouse clicking above a figure, is dynamically created. Depending on the type of the entity which is underlying a displayed figure as model, the right plug-in hierarchy is scanned for applicable commands to this specific entity. Like that, no unnecessary menu entries distract the user's attention.

The main window implements all the tool requirements *Pushing Information*, *Pulling Information*, *Near Features*, and *Automation* which we identified as addressing the concern *Efficiency*.

Session Viewer

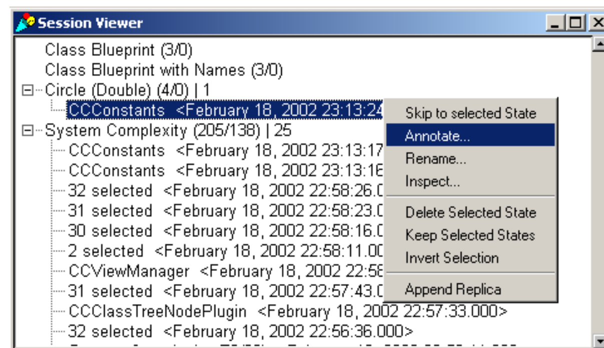


Figure 5.2: Screen Capture of *MooseNavigator* Session Viewer.

Main purpose of the session viewer is to manage the tool states within a session, and giving an overview of previous views and selected entities. The context menu offers possibilities for renaming, deleting, annotating, and inspecting session states. Selections of multiple states can be removed, or inverted. Single states can be cloned and appended to the session. States are displayed in a tree, grouped by view type, after that sorted by time of creation. The session viewer user interface is shown in Figure 5.2.

The session viewer primarily addresses the concern *Navigation between Tool States* but also implements some tool requirements addressing *Efficiency*, *Feedback*, *Consistency*, and *Memory*.

Description Viewer

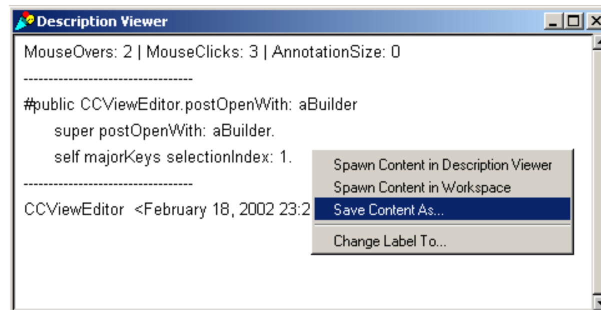


Figure 5.3: Screen Capture of *MooseNavigator* Description Viewer.

Figure 5.3 shows the description viewer user interface. Main purpose of the description viewer is to continuously display as much as possible of the available detailed information about the current item with focus. This includes a verbose description of the view, if no specific figure has the focus; a detailed description containing the name, characteristics, annotations, and metric values, if the figure's model is a class, attribute, inheritance definition or an access; and additionally the source code, if for example a method-figure has the focus. A special context menu allows the user to change the label of the window; save the current text to a text file; spawn the content in another description viewer; or spawn the content in a workspace.

The description viewer implements *Pushing Information*, a tool requirement addressing the concern *Efficiency*.

System Overviewer

Figure 5.4 shows the system overviewer user interface. Main purpose of this small window is to provide a permanent overview of the current view, the actually visible area and selected figures. The drawing contains three visual elements:

1. A black frame. It shows the closure of all nodes in the current view.
2. A bold red frame. It shows the currently visible area within the current view.
3. Small black circles. They symbolize the center position of currently selected nodes and edges.

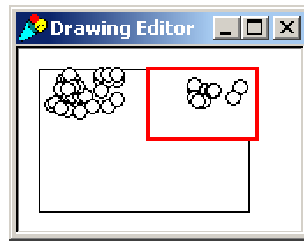


Figure 5.4: Screen Capture of *MooseNavigator* System Overview.

The system overviewer implements *geographical consistency* which is introduced in Section 4.3. In general it addresses concerns of *Complexity Reduction* and *Consistency*.

Filter Editor

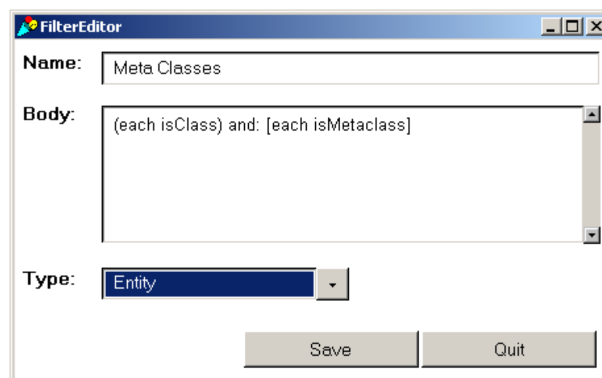


Figure 5.5: Screen Capture of *MooseNavigator* Filter Editor.

Figure 5.5 shows the filter editor user interface. Main purpose of the filter editor is to define filters. Filters are used for showing or hiding specific content in the views, e.g., in situation where a certain aspect is not relevant. The three main input fields for declaring a filter are:

1. Name. This is the display name for the filter.

2. Block. This must be a valid condition for a Smalltalk block, whereas *each* is the input parameter and has the role of the current entity, when the filter is applied.

3. Type. This is the type of entity to which a filter shall be applied, or not. If no specific type is selected the filter has general type *Entity*. Possible types are: *Class*, *Attribute*, or *Method*.

Filtering explicitly implements the tool requirement *Reducing Volume* as one of the requirements addressing *Complexity Reduction*.

Filter Library Editor

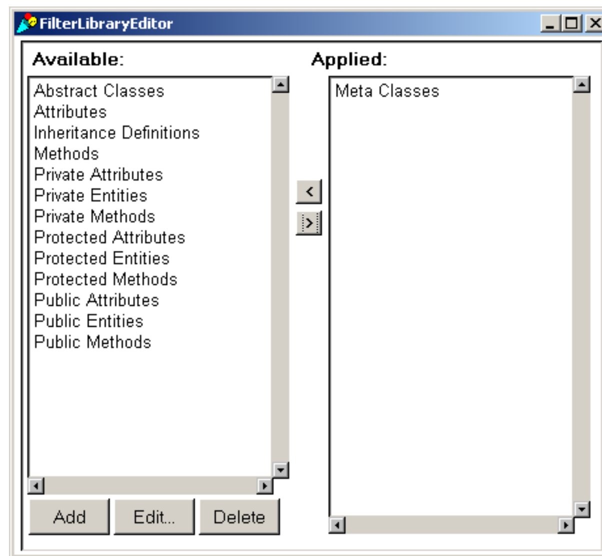


Figure 5.6: Screen Capture of *MooseNavigator* Filter Library Editor.

Figure 5.6 shows the filter library editor user interface. The main purpose of the filter library editor is to manage the filter library, but also to define what filters currently shall be applied. This is the place to add, remove, or edit existing filters. The repository of filters is kept centralized and only once in the system, whereas the configuration regarding which filters are currently applied and which are not, can be specified for every single session state.

5.2.3 Metrics

For being able to access navigation tracking information efficiently, we introduce two new metrics, which can be applied to any kind of entity:

> **NTMC.** *Navigation Tracking Mouse Clicks.* This is the number of mouse clicks that were performed on a figure, representing an entity.

> **NTMO.** *Navigation Tracking Mouse Overs.* This is the number of times that the mouse moved over a figure, while in the meanwhile having again been moved over another figure.

With these new metrics we have an instrument at hand for measuring the navigation intensity in a model. These metrics are a prerequisite for the navigation trails which are presented subsequently.

5.2.4 Layouts & Views

Circle Layout. Sometimes the metaphor of a cherry is used to illustrate encapsulation in an object-oriented class. The core is the class description, the meat is the data and internal behavior, and the peel the publicly accessible interface. With this analogy in mind we wanted to have a layout at hand, which allows us to layout sets of entities in circles. Since in *CodeCrawler* there was already a circle layout we used this and extended it. This resulted in a circle layout which arranges a set of entities around a centered entity. The layout can be used recursively to obtain multiple layers. We provide the possibility to layout nodes in a circle with a fixed size radius, or a radius that is computed for fitting in the current size of the drawing window.

The circle views implement views to show the class internals which in the previous section was identified as one of the two most popular and efficient kinds of views. At the time of its origin the *Class Blueprint* was not available yet.

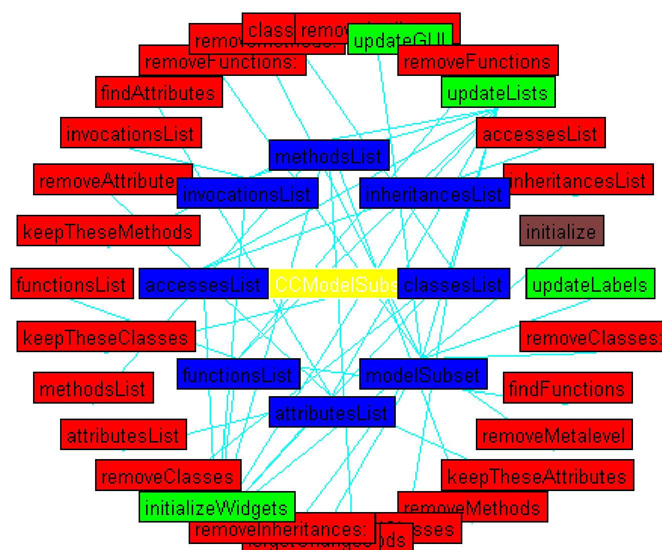


Figure 5.7: A Typical Circle (Double) View.

Circle (Single). With the above analogy to a cherry we created new views to present the class internals. The simplest view positions the class entity in the center of the drawing, and arranges around it, all methods and attributes. The edges are drawn along accesses of methods to attributes.

Circle (Double). This view is like Circle (Single) but draws two circles. The set of entities to arrange in each circle can be passed as parameter list. Usually

we draw the attributes in the inner circle, and the methods in the outer - with the ideas of encapsulation in mind. Figure 5.7 shows an example view. In the middle is the class entity in yellow, in the inner circle the attributes in blue, and in the outer circle the methods. Methods are colored according to their characteristics: Green for initializers, gray for constants, brown for extenders, or red if not further specifiable. Additionally nodes with $NTMC > 0$ show their name in white, instead of black, to directly demonstrate that they already have been visited.

Circle (FixedSize). This view works like Circle (Double), but has a fixed radius size, so that the circle has always the right size to the number of contained entities. The trade-off between Circle (FixedSize) vs. Circle (Double) is the risk of needing to scroll vs. the risk of overlapping nodes.

Navigation Trails. The Navigation Trails view draws blue edges along navigation paths. Like that, you can see from which node to what node you have navigated. The color metric of nodes stays like it is in system complexity (WLOC), width and height are bound to the new introduced metrics (NTMC and NTMO respectively). Figure 5.8 shows an extract of a system complexity view (upper left), which over time gets navigated (below the same extract evolving at three different times). With this kind of view you may identify heavily navigated parts of the system, and others, that were not considered in previous investigations. Entities which only attract users to mouse over get slim and tall, entities which the user has a closer look at it terms of clicking on the figure get rather fat.

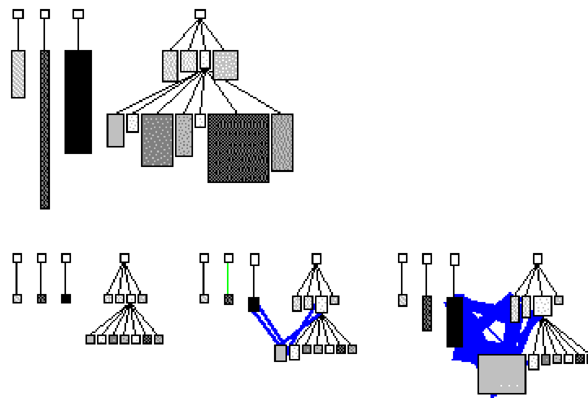


Figure 5.8: Typical Navigation Trails Evolving over Time.

Navigation trails are a concept that address the concerns *Feedback*, *Classification*, and *Memory*.

5.2.5 Reports

The user session can be asked for several reports. The tab-separated columns can easily be used in any other data analysis or data mining tools. *Reports* are a major tool requirement addressing the *Storage* concern. Here are some predefined reports:

Report Visited Entities / Report Visited Classes. This report is a simple list of the names of all visited entities, or classes respectively:

```
ArborModuleComponent
MenuItem_class
Collection_class
ArborModule
ArborCoreUtils
ArborRedefinedClassSignature
ArborAbstractTree
ArborModuleComponentReader
```

Report Actions and States. This report verbosely informs about past user interactions. This first line reports when the session has been created. The following lines report actions and states. For actions the columns are: timestamp, name, receiver, message. If it is a state the columns are: timestamp, name, numberOfEntities in the current view, view name, view nodes size, view edges size, filters size, first uuid in selections, first name in selections, annotation:

```
Session: MooseNavigator Session, started on: February 16, 2002 23:05:19.000
February 16, 2002 23:05:27.000 'System Complexity (205/138)'14739 'System Complexity'0 0 0
February 16, 2002 23:05:28.000 'MooseNavigator.crawlWithView:'MooseNavigator #crawlWithView:
February 16, 2002 23:05:28.000 'MNDrawing.selectNodes:'MNDrawing #selectNodes:
February 16, 2002 23:05:28.000 'MNDrawing.raiseNodes'MNDrawing #raiseNodes
February 16, 2002 23:07:38.000 'MNDrawing.redButtonPressedOnDrawing'MNDrawing #redButtonPressed
February 16, 2002 23:07:38.000 'System Complexity (78/69)'4612 'System Complexity'0 0 0 137 #Model
```

Report Annotated Entities. This report lists all entities for which an annotation has been created:

```
#chkFunction -> This method smells - duplicated code
#CCMetricSubcanvas -> Should be split!
#CCMetricSubcanvas_class -> This class is never used
```

Chapter 6

Experiments

"Example isn't another way to teach, it's the only way to teach."

-Albert Einstein, physicist

In search of ways for dynamically providing features for navigation, we did an experiment of automatically extracting possibilities of navigation from a specific meta model description. We describe why we found this not to be a promising method in Section 6.1.

To validate the statements of the previous chapters about the user behavior in reverse engineering, we set up two experiments of letting people work with *MooseNavigator* in real reverse engineering sessions. During the sessions we recorded the behavior of the users. Section 6.2 and Section 6.3 present the analysis of the collected data and experiences.

6.1 Automatic Navigation Support

In search of ways for automatically computing all possible navigation steps on a certain model, one of our ideas was to scan the meta model for all its relationships (see also [FAVR 01]). The goal was to use the set of relationships as a base for possible navigation steps from one topic to related topics. If this would be possible we could use the information for automatically and dynamically create menu entries and navigation features. No human was necessary to decide in tool design time which navigation steps to support, or what navigation paths to allow.

In the following we describe our proceeding by giving a concrete example. Our first attempt was to check paths to neighbor topics along associations. Our meta model under consideration is *Moose* (Appendix A), the Smalltalk implementation of FAMIX, described in Section 2.2. With the assumption that *Moose* is complete, all the needed information for our endeavor is available. We purposely start with

Class Hierarchy	*Number of Protocols*	*Number of Methods*	*Accumulated Number of Methods (without Object's)*
MSEAttribute	8	22	**155**
MSEAbstractStructuralEntity	8	40	133
MSEAbstractEntity	4	10	93
MSEAbstractObject	7	33	83
MSEAbstractModelRoot	7	39	50
MSEAbstractRoot	4	11	11
Object	30	181	(0)

Table 6.1: Methods of MSEAttribute's Fullclass.

a non-technical example, with the aim of reducing the risk of confusion between objects (attribute) and meta objects (*MSEAttribute*).

> **Example:** We consider an object-oriented class ***Wine*** which has an attribute ***vintage***.

In *Moose*, classes are represented by instances of *MSEClass*, and attributes are represented by instances of *MSEAttribute*. If we now focus on this *MSEAttribute*-instance *vintage*, and if we want to find all possibilities of relations from such an entity, we can collect all attributes and methods of *MSEAttribute* - since this would be all the explicit information we have in the model. By doing that, we obtain all direct neighbor topics of *vintage*. In a first step we focus on the methods of *MSEAttribute*.

As (Table 6.1) shows, the total number of methods of *MSEAttribute* and all its superclasses is 155. This is greater than a reasonable number of navigation options. On the one hand, the methods of the Smalltalk root class *Object* were not counted. On the other hand, there might be counted a little bit too many methods, since abstract methods, overridden by subclasses, were counted multiply. However, this imprecision falsifies the real number of available methods for a class only marginally. Still, the set of methods is too big to serve as base for proposed navigation steps. We could try to reduce the number of candidate navigations by focusing on a certain kind of methods, like public methods, private methods, initializers, et cetera. None of these approaches seems promising to us, because they are not restricting enough or hard and vague to find out.

After having discussed navigation along the methods of *MSEAttribute*, we now focus on the attributes. We realize that there are not as many of them as for the methods - we can list them all explicitly, as done in Table 6.2. Before we are able to explain this table we need to introduce two more concepts.

> **Definition: A *domain class*** is a class that has its origin in the field of the current project-specific considerations. In our case typical *domain classes* include *MSEClass* and *MSEAttribute*.

Class Hierarchy	*Attribute*	*Class of Attribute's Value*	*Domain Class*	*Reasonable Navigation*
MSEAttribute	**belongsToClass**	MSEClass	Yes	Yes
	accessControlQualifier	String	No	No
	hasClassScope	Boolean	No	No
MSEAbstractStructuralEntity	**declaredType**	(Type of Represented Entity)	(Yes)	(Yes)
	declaredClass	(Class of Represented Entity)	(Yes)	(Yes)
	interfaceSignatureSet	Collection of Strings	No	No
	accessedByList	Collection of MSEAccess	(Yes)	(Yes)
MSEAbstractEntity	name	String	No	No
MSEAbstractObject	**namedPropertiesDict**	Collection of anything	(Yes)	(Yes)
	commentCollection	Collection of Strings	No	No
	sourceAnchor	String	No	No
MSEAbstractModelRoot	uuid	Integer	No	No
MSEAbstractRoot	-	-	-	-
Object	(-)	(-)	(-)	(-)

Table 6.2: Attributes of MSEAttribute's Fullclass.

> **Definition:** ***Non-domain classes*** are all the classes that are not *domain class*. These include base classes like *Integer, Collection*, or *String*.

We come back to our previous example. An example *Wine* might be of the year 1981. This means that the attribute *vintage* holds an instance of the class *Integer* with the value "1981". We would say that *vintage* holds instances of *non-domain classes*. In contrast, if our class *Wine* would have another attribute *chateau* holding instances of *Chateau*, the latter would be called *domain class*.

What we observed now, was a relationship between the reasonability of navigating along a certain attribute and the fact whether this attributes hold instances of *domain classes*.

Here is how we came up to this speculation. To explain it, we reconstruct the content of Table 6.2 step-by-step. We start with columns 1 & 2, which list *MSEAttribute* and all its superclasses, each with all its attributes. Again we do not

consider the Smalltalk root class *Object*, since its attributes have nothing to do with the semantics of our example.

- The class *MSEAttribute* has an attribute *belongsToClass*.

Now we add column 5 where we put our estimation for the reasonability of navigating along each attribute.

- We find it reasonable to navigate from an attribute to its class. Example: Navigate from *chateau* to *Wine*.
- We find it **not** reasonable to navigate from an attribute to its access control qualifier. Example: Navigate from a *chateau* to *"Private"*.

If we look at the table and try to derive a rule we find that those attributes that are not reasonable for navigation usually hold some kind of *"leaf-information"* which does not offer ways of further navigation. Examples of *"leaf-information"* include "Private", "TRUE", or "129748". The classes of these values are Strings, Booleans, and Integers - all representatives of *non-domain classes*. We add columns 3 & 4 and indeed observe a correlation between the values in the columns *Domain Class* and *Reasonable Navigation*.

So far our view was restricted to direct one-to-one neighborhood along associations between topics. Now we widen our angle of view, and inspect more complex relations, based on compositions of associations.

> **Example:** To find ***composed relations*** like a ***method's overriders*** all the method's class' subclasses must be parsed on methods that extend this particular method.

This indirect neighborhood can be important, especially in cases of complex relations which are not explicitly modeled by direct associations, like in the given example. Another case of composed relations, is the occurrence of transitive relations between topics, like *"is_superclass_of"* between two classes. However the potential set of all compositions of associations between topics is huge. Since all these compositions have primarily the same possibility of being useful, we need a way to reduce the set, like a concept of priority.

With the concept of affinity we have an instrument to measure the relevance of these relations. To be able to determine affinity we must assign weights to associations. For that we need information about the semantics of the model, and we need also to know the specific task which has to be performed. When we do a change-impact-analysis, affinity between topics must be different from affinity, when we search duplicated code. The configuration of affinity can not be done automatically. There are two ways for favoring associations: First, we know some

important relations, because we use them every day, like extenders of a method. Secondly, we can track the user's behavior, collect information about the paths, identify so called trails, and try to abstract the importance of indirect associations with the help of data mining and statistics.

Summary: It is difficult to find useful navigation support computationally. Knowledge about the semantics of the model is needed to introduce concepts of affinity and relevance. Finally, human experience is inevitable to define reasonable navigation support. Navigation tracking can help to reason about the users' behavior, and to improve the efficiency of a user interface.

6.2 Experiment with Students

Within the scope of the practical lecture *Software Engineering Applied*, the students of the University of Bern learn about reverse engineering. In a lab session of two hours they used *MooseNavigator* to exercise the use of reverse engineering tools to examine an existing object-oriented system. The subject system was *Moose*, including *CodeCrawler*. After a short introduction, four groups of three to four people sat together, and with the help of three assistants, tried to get an idea about the subject system.

6.2.1 Observations

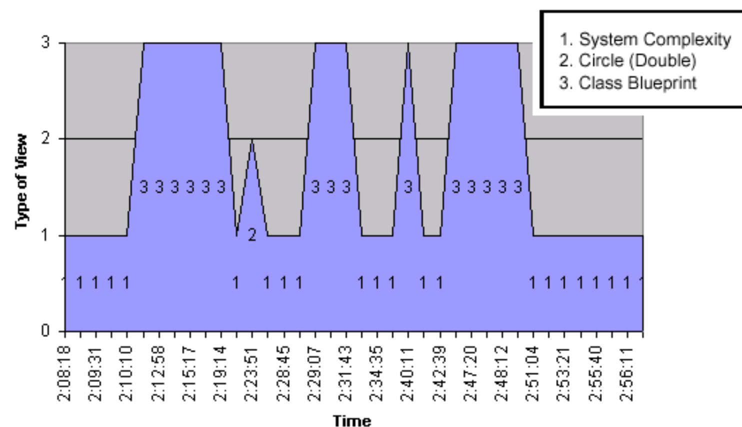


Figure 6.1: Procedure of a Typical Student User Session.

Users surf always either on top or on the bottom of the abstraction depth. This is typically at the level of an inheritance tree, or down at the detail level of class internals and in the methods' bodies. Figure 6.1 shows a typical reverse engineering session. In the horizontal dimension are the different types of views the user selects and the vertical axis covers the time that passes by staying at the particular

User Interaction	*Number of Occurrences*	*Average Time in between [in Minutes and Seconds]*
Change Selection	38	1'16"
Change Filter	-	-
Dive	8	6'00"
Change View Type	10	4'48"
Pop	5	9'36"
Spawn	-	-

Table 6.3: Average Time between Student User Interactions.

state. The particular views selected in the session are *System Complexity, Circle (Double)*, and *Class Blueprint*. Examples of these views can be found in the following figures: Figure A.3, Figure 5.7, and Figure 3.3.

Users interact seldom. We were astonished how seldom users change the view and how long they tend to stay on one certain view. Table 6.3 shows the average time that passes between the user's interactions. Basis of these measurements is the above user session. The observed period was 48 minutes - depending on what a user wants to find out, and depending on how much expertise he already has about the subject system as well as in using the tool, the numbers probably vary a lot. Some of the basic features were not used at all. The corresponding fields in the table are marked by "-".

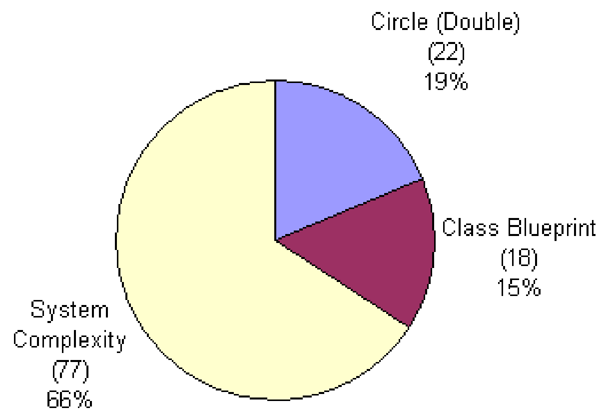


Figure 6.2: Distribution of Selected Views by Students.

Users select always the same few views. Figure 6.2 shows the only three different kinds of views which are selected over the complete time. This is probably because users, once familiar with a certain kind of view, feel familiar and produc-

tive with it, and do not want to get confused by introducing new views, and with it new aspects.

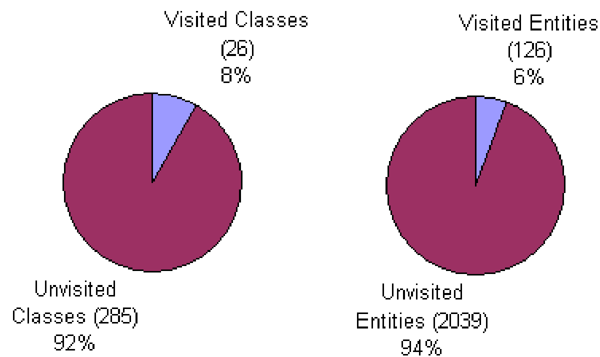


Figure 6.3: Percentage of Visited Classes and Entities by Students.

Users surf vertically and horizontally, but not diagonally. Figure 6.3 shows the relative coverage of visited classes or entities during a session. The real number of entities is 39,462 which covers approximately 10,000 accessors, 8,000 accessor arguments, and 6,000 expression arguments. In our analysis we considered only the most important entities, which also are represented as nodes in the graphs. These are classes, attributes, and methods. The number of these entities is 2,165. Accessors are implicitly visited when we follow edges in the graph. We did not explicitly count these visits, because we believe that the user did not want to visit the accessor itself, but the user only uses the relation to find a related target. The correlation between the ratio of visited classes within the total number of classes, and the ratio of visited entities within the total number of entities, can be interpreted as a tendency to the fact that users either navigate just on the top level of abstraction, i.e., on hierarchy diagrams of classes, or they navigate at the bottom level of detail, i.e., in methods source code.

Users complain about missing features. By, for example, scrolling a view in *MooseNavigator*, the *System Overviewer* is automatically updated. Users expected to be able to move the squares in the *System Overviewer* with the feedback-effect of really changing the current visible part of the system. This functionality was not provided by the prototype. Users valued this to be disturbing, inconsistent and incomplete.

6.2.2 Experience

In the beginning we thought that the results will show a broad diversity, which means that many users show much different behavior. Many outer circumstances influence a reverse engineering session. A pair of engineers quickly gets to dis-

cussing about patterns and anti-patterns they recognize, good or bad style of coding et cetera. They talk about questions like why this node is colored green, and whether it does get the color of being an initializer, or of being an extender, if it is satisfying both criteria. The reasons that people actually do something other than what they currently are expected to do are manifold. We have to consider that the views presented by our tools are complex, so that users need a lot of time to digest what they see on the screen, before they click further. The behavior of different teams are quite similar, we suppose that the relevant dimensions that influence a session were similar in our case. They are: Subject system, given task, instructions of supervisor, expertise of the user with a) the subject system, and b) the tool.

6.2.3 Results

We summarize the main results of the experiment with students as follows:

- Different users show similar behavior.
- Users like to navigate either on top or at the bottom levels of abstraction.
- Users select always the same few views.
- Users interact seldom, since they need a lot of time to understand a single view.
- Users prefer vertical and horizontal to diagonal navigation.
- Users complain about not consistent or incomplete functionality.

Since the observed users did not know the system, since they were beginners in reverse engineering, and since they were not used to work with *MooseNavigator*, they might behave different from experienced users. We supposed professional reverse engineers to surf faster than novices, since they can quicker interpret and understand what they see. For verifying this speculation we set up another experiment of observing an expert user. This experiment is described in the next section.

6.3 Experiment with an Expert

Duploc is a tool for the visualization of duplicated code. It is written in Smalltalk, and was developed at the University of Bern[1]. For about one hour, the system has been reverse engineered by an expert of *CodeCrawler* using *MooseNavigator*. His aim was to detect the internal architecture. The expert did already have an idea about the functionality of *Duploc* and its design. However, he did not yet have a closer look at the project and its source code before.

[1] http://iamwww.unibe.ch/~rieger/duploc/

Loading *Duploc* into *Moose* leads to the following picture: 758 classes; 5,493 methods; 794 attributes; 634 inheritance definitions; 19,327 invocations; 30,682 accesses; with a total number of entities of $\sim$ 95,000. Since *Duploc* is written in Smalltalk where a meta class exists for each class, and also the necessary base classes were extracted by the parsers, we estimate the number of "real" classes in the system to approximately 300.

6.3.1 Observations

The expert utilizes eight different views while inspecting the system. This is almost three time as many as the a typical student utilized in the same time. Figure 6.4 shows the procedure of visited views during the expert session. Most of the transitions from one view to the next view are of a *horizontal* navigation nature, like changing from a *System Complexity* view to another *System Complexity* view while only changing the selection. The *dimensions* of navigation are described in Section 4.3. All horizontal transitions in Figure 6.4 correspond to *horizontal* navigation steps, though changing the view type can be a *vertical* navigation step when keeping the same selection, or a *diagonal* navigation step when changing view and selection.

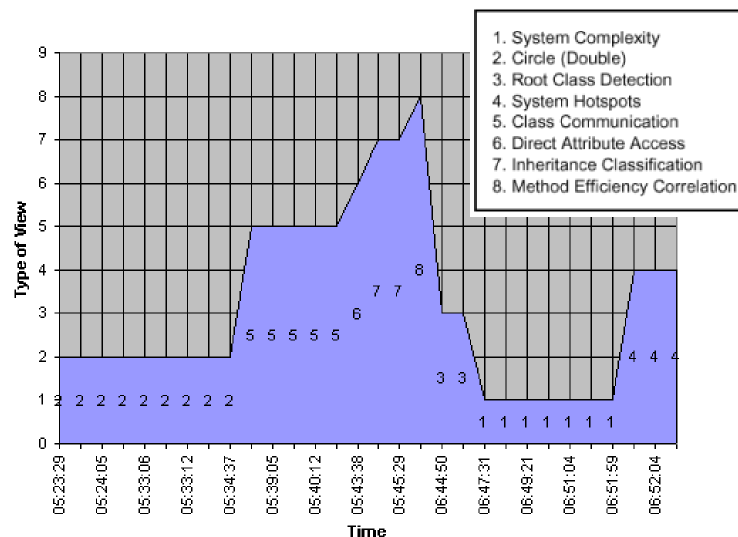


Figure 6.4: Procedure of an Expert User Session.

Only few *vertical* navigation steps are performed, like diving from *System Complexity* to *Circle (Double)* which is beside the *Class Blueprint* another view to show the details of a class' internals. A detailed description of these views can be found in Section 5.2 and [LANZ 99]. Which view is visited how many times is shown in Figure 6.5.

User Interaction	*Number of Occurrences*	*Average Time in between [in Minutes and Seconds]*	
		Expert User	*Students*
Change Selection	59	0’57”	*1’16”*
Zoom	6	9’30”	-
Change Filter	2	28’30”	-
Dive	4	14’15”	*6’00”*
Change View Type	6	9’30”	*4’48”*
Pop	4	14’15”	*9’36”*
Spawn	-	-	-

Table 6.4: Average Time between Expert User Interactions.

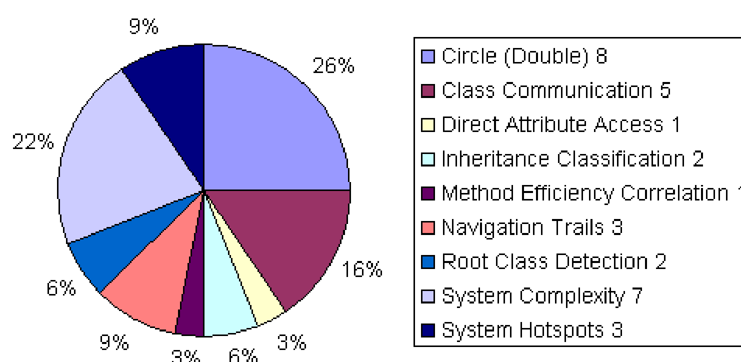


Figure 6.5: Distribution of Selected Views by an Expert.

In contrast to our speculation - the expert does **not** navigate faster than the students. Also the expert pauses an average of around one minute after the creation of a view, before he performs his next action. Table 6.4 shows a comparison of performed user interactions by the expert and by the students.

Also the expert uses many features either seldom or never. For example the expert never renames a tool state so that he could more easily remember what it shows.

Albeit seldom, in contrast to the students, the expert **did** use most of the advanced features like zooming, filtering, selecting nodes matching a certain criteria, applying advanced layouts to a selection of nodes, coloring a selection of nodes, skipping among tool states, or annotating tool states.

A class counts as visited when a user clicks on the node, e.g., for diving into the details. The total number of visited classes in the session is only 13. As illustrated in Figure 6.6 this is around 2% of the system. Considering the number of visited entities, we get a coverage of visited entities among the whole system of

0.002%. This is a sign that the expert did not browse into the details of the system, but surfed primarily on the top abstraction level, where the nodes represent classes, and the views make statements about the overall architecture rather that about the class' internals.

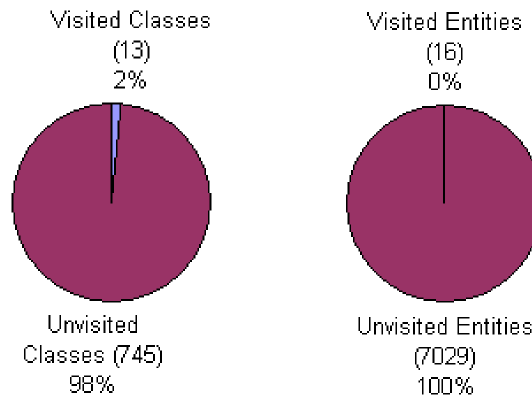


Figure 6.6: Percentage of Visited Classes and Entities by an Expert.

6.3.2 Experience

In an interview the expert describes the procedure as follows. The expert starts with a *System Complexity* view to get an overview of the system. By removing the meta classes he reduces the number of nodes by a factor two. Three similar trees let the expert suppose a concept like Model-View-Controller (MVC) [CINC 02] to be utilized in the design of the system. After moving the mouse over the corresponding trees, and reading the names of the classes, this speculation seems to be verified. Apparently *Duploc* implements a lot of Graphical User Interface (GUI) specific functionality by itself, rather than using a third party framework. The rest of the system is not characterized by deep inheritance, the class hierarchy is rather flat. Still on the same view the expert proceeded by coloring nodes according to their characteristics, e.g., *abstract* classes in yellow, GUI classes in red, and the rest in blue. This lead to the speculation that approximately one third of the system can be considered as *domain*. The standalone classes have to be visited separately.

Advanced grouping mechanisms would have been helpful to classify nodes, and to have and additional criteria for coloring or arranging nodes. Advanced grouping mechanisms include also aggregation. The fact that detailed information is displayed automatically in the *Description Viewer* while moving the mouse over an object, reduces the necessity of diving into the details. This increases the efficiency in inspecting a number of classes one after the other.

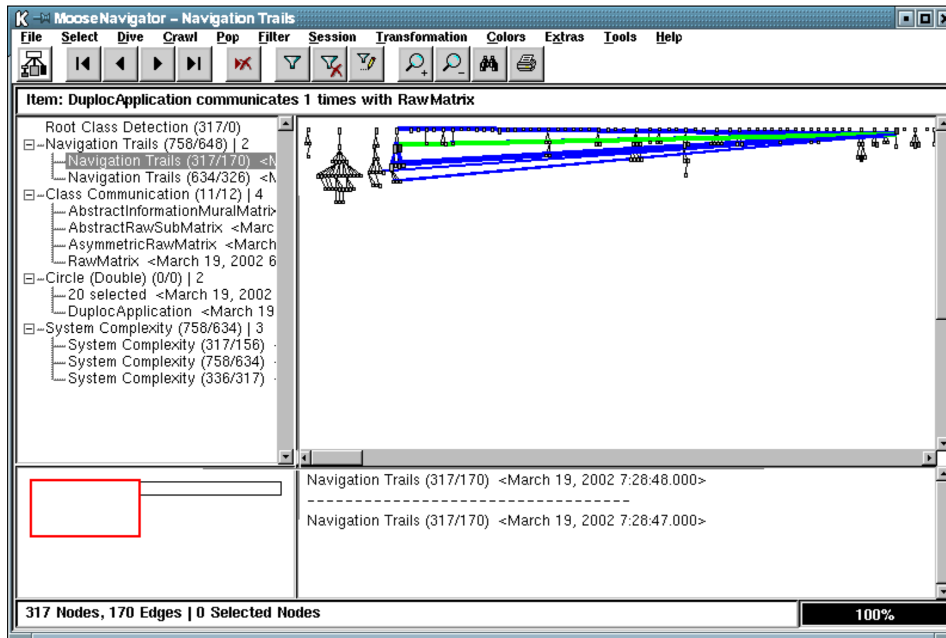


Figure 6.7: Navigation Trails in the Duploc Experiment.

After the session, we wanted to visualize what the expert visited. For that we created a *Navigation Trails* view, shown in the screen capture Figure 6.7. Outstanding is that all the navigation trails end in the same place. It is the class *DuplocApplication* from which most of the functionality seems to be controlled. The view shows only navigation trails between classes, which have been selected explicitly (e.g., by diving down to the class internals), and not between classes which the expert only inspected by moving the mouse over the node and by reading the description of the class in the *Description Viewer*. Since also annotations for each artifact are displayed, the expert annotates views and classes with his insights and his experiences. Finally, the expert regards the possibility to navigate among tool states, and being able to skip back and forward in the list of views.

6.3.3 Results

We summarize the main results of the experiment with an expert as follows:

- The expert utilizes more different views.
- The expert filters information before considering it in detail.
- The expert primarily navigates *horizontally*.
- The expert primarily navigates on the top level of abstraction, and seldom navigates into the details.

- The expert does not navigate faster that the novices.
- Also the expert uses many features seldom or never, however he used most of them at least once.
- The expert visits only a small fraction of the system.
- By visualizing class hierarchies the user can detect concepts like MVC.
- With the help of identifying concepts like MVC the user can separate interface and domain.
- By applying metrics to visualizations of class hierarchies the expert can detect hot spots.
- The expert explores the system starting by inspecting hot spots like god classes [BROW 98].
- The expert misses grouping mechanisms.
- The expert regards the possibility to see detailed information by moving the mouse over nodes and edges.
- The expert regards the possibility to navigate among the set of previous views.

6.4 Summary

Automatic navigation support is difficult. Since the number of artifacts and relations in object-oriented software keep rather stable, we suggest a static selection of features, based on human experience and the analysis of tracked user behavior.

In an experiment with students we observed in detail the behavior of users. We identified many users to behave quite similar. They use few features and interact seldom with the tool.

In another experiment with an expert user we disproved our speculation that he would navigate faster then the novices. However, the expert utilized more different features.

We would like to make more experiments and case studies. To date this was not possible due to limited time resources.

Chapter 7

Conclusion

"Selbstverständlichkeit existiert nicht."[1]

-Gilbert Probst [PROB 99]

7.1 Summary

Many issues of navigation in object-oriented reverse engineering are the same in knowledge management. Two major problems are complexity and inefficiency. We picked up further similarities and differences of both areas, with respect to managing, modeling, and navigating information. We studied the factors for success of a reverse engineering navigator which resulted in a set of concerns. This set was obtained by accumulating and bringing together the following information and insights:

- Ways of reducing friction in the reverse engineering process, addressing its major causes, among which are the incompleteness of the model, missing features, indirection of paths, oversaturation of information, red herrings, degradation of knowledge, and the lack of mechanisms for classifying information and relations (Chapter 1).
- Requirements for supporting the tasks of knowledge management: Identify, create, collect, filter, categorize, store, distribute, utilize, and finally maintain knowledge (Table 2.1).
- Key success factors for knowledge management tools, which are simplicity, maintainability, user-friendliness, low political, psychological and structural barriers (Table 2.2).
- Formal needs derived from paradigms of HCI, like simplicity, efficiency, memory, consistency in state an orientation, appropriate features, and extensibility (Section 2.3).

[1]Literally: "There is not a single matter of course."

With this set of concerns we had a base for further considerations and a structure for identifying strengths and weaknesses in our own reengineering environment *Moose* (Chapter 3). We compiled a list of concrete requirements for navigation support in reverse engineering tools by merging formal needs with the experience of an industrial case study (Section 3.3). We analyzed also best practices and features of state-of-the-art navigation (Chapter 4). We put all this information and experience again together in Chapter 5, and came up with the following perceptions:

- Users of reverse engineering tools are slow in generating new views. But they navigate faster in the list of previous views and along the level of detail.
- The more expertise a user has the more features he uses, still many features are seldom used.
- Efficiency in navigation first depends on the support for not getting lost, second on views that reduce complexity, and third on appropriate features to find relevant information.
- While navigating through a system users prefer to change only one dimension of a view at the time, otherwise it seems to be too hard to follow.
- Extensibility is a key feature for a tool that assists us in doing such complex and multifaceted tasks as reverse engineering a software system.

Many of the above statements could be validated or even have its origin in the described experiments (Chapter 6), and by a proof of concept, in the form of a prototype navigator for object-oriented reverse engineering (*MooseNavigator*, which is described in Section 5.2).

7.2 Main Contribution

1. This work outlines the involved theory and issues of building useful reverse engineering tools. It proposes a vocabulary and a simple taxonomy for the issues of managing, modeling and navigating models of object-oriented systems. It results in a set of concerns for an ideal navigation tool in reverse engineering.
2. We compile a set of requirements, concrete solutions and examples, of ways how to implement and address the before identified concerns.
3. State-of-the-art navigation tools - together with our own prototype - are compared and measured at this list of concerns. This lets us identify:
   (a) Best Practices & Solutions.
   (b) Gaps, Problems & Challenges.

4. In an analysis we classify and discuss navigation steps - the building blocks of navigation - and provide detailed lists of representatives.

5. Finally we come up the some perceptions that can help to value and distinguish different issues of navigation in reverse engineering. We identify some of the major sources of friction and also show solutions of how to reduce this unnecessary extra effort.

7.3 Outlook & Future Work

Prototype integration. Our experience and feedback shall be used to enrich the *Moose* reengineering environment. A merger of *CodeCrawler* and approved ideas, features, and views of *MooseNavigator* is the first step.

Extending *Moose*. *Moose* will be extended towards enhanced classification capabilities, aggregation, a distributed version of the shared repository. Round trip engineering could be another goal. A command repository could formalize queries, test prerequisites, and the applicability according to the set of parameters. There are no limits to the creativity in designing new, different layouts and views. Better support for directly illustrating parts, packages of a system would be helpful. More information about the runtime behavior of the entities in the system could lead to new insights.

Meta meta models. In the field of meta meta models, still a lot of work has to be done. The semantic web, or topic maps are concepts in the direction of a general system for representing any kind of information. The future will show, whether these standards will be established, or not. In any case some sort of ontology is needed, for being able to dynamically add new kinds of entities to the current model, and to build bridges to other information systems.

We would like to make more experiments and case studies, to have a broader validation, and to get additional input for understanding the behavior and needs of reverse engineers.

Appendix A

Moose

In the past few years the Software Composition Group (SCG) at the University of Bern was involved in a number of research projects in the field of software re- and reverse engineering. In the FAMOOS project leading European partners came together to build a number of tool prototypes to support object oriented reengineering. FAMOOS is an acronym for **F***ramework-based* **A***pproach for* **M***astering* **O***bject*-**O***riented* **S***oftware* which is the name of the ESPRIT project 21975, a research and development sponsorship programme of the European Union on information technology. The three year project ended in September 1999. More on that project can be found online[1].

The above mentioned prototypes were validated during experiments on various case studies. The source code of the available case studies was written in different implementation languages (C++, Ada, Java and Smalltalk).

To avoid equipping the tool prototypes with parsing technology for all those programming languages, a common information exchange model with language specific extensions was specified (see Figure A.2). This model has been named FAMIX (**FAM***OOS* **I***nformation E***X***change Model*). Another practical result of the FAMOOS project is the "FAMOOS Object-Oriented Re-engineering Handbook" [Duc 99]. It collects techniques and knowledge on the problem of software evolution with a special emphasis on object-oriented software. Most of the subject matter is not "new" in the sense that it represents new discoveries. Rather the handbook regroups much of the knowledge about redesign, metrics and heuristics into a single work that is focused on practical issues in object-oriented reengineering.

Since the end of the FAMOOS project in 1999 we further evolved the tools and optimized the model. In the following we present the meta model and the current state of the most important tools.

[1] http://www.iam.unibe.ch/~famoos/

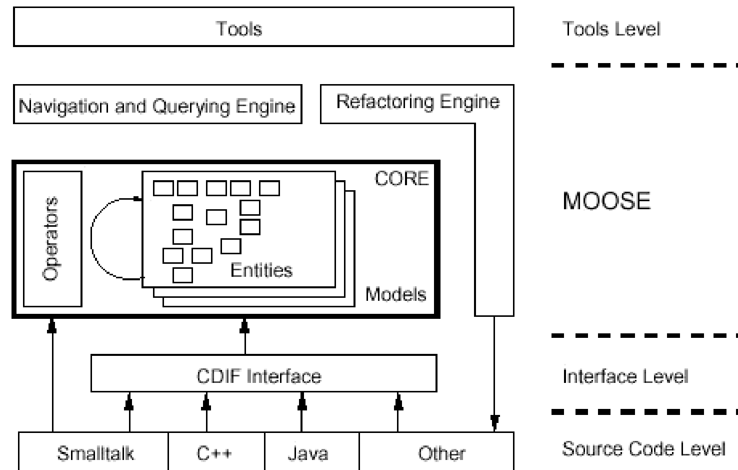


Figure A.1: *Moose* Architecture.

Moose is our reengineering research platform implemented in VisualWorks Smalltalk [DUCA 00a] [DUCA 01b] [TICH 01]. It has been developed during the FAMOOS project to reverse engineer and re-engineer object-oriented systems. It consists of a repository to store models of source code. The models are stored based on the entities defined in FAMIX. The software analysis functionality of *Moose* is language independent. The FAMIX models can be loaded from and stored to files. Apart from the repository, there are other features implemented to support reverse engineering activities:

- a parser for Smalltalk code
- an interface to load and store information exchange files
- a software metrics calculation engine
- an interface for additional tools to browse and visualize stored entities

A.1 Meta Model

The FAMIX core model (Figure A.2) consists of the basic entities in object oriented languages, namely Class, Method, Attribute and InheritanceDefinition [DEME 01] [TICH 01]. For reengineering we additionally need to know about relations between the basic entities. Invocations and accesses provide information about such relations. An Invocation represents the definition of a method calling another method. An access represents a method accessing an attribute. These abstractions are needed for reengineering tasks such as dependency analysis, metrics computation and reengineering operations.

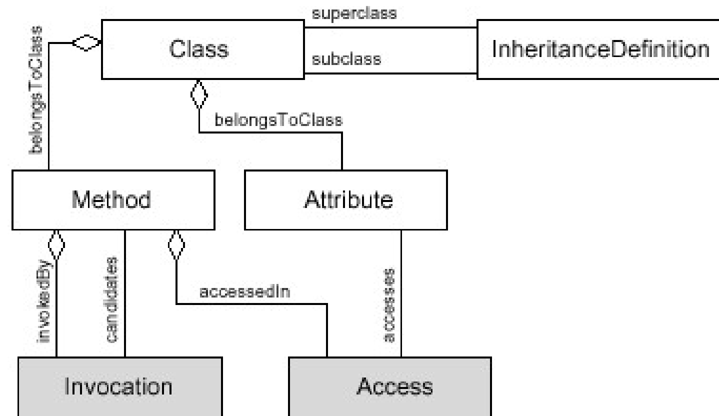


Figure A.2: FAMIX Core.

To satisfy the need for information exchange between tools, the CDIF standard was chosen in the FAMOOS project as the basis for transferring information. CDIF is an extensible format supported by industry standards. The plain text encoding facilities of CDIF have ben adopted to support information exchange between tools. The chosen format is human readable and simple to process. The need for data exchange has increased rapidly in the last years through the wide use of the Internet. XMI has been accepted in industry as a new standard for information exchange. We plan to shift from CDIF to XMI as exchange format to keep compatibility with industry standards.

A.2 CodeCrawler

CodeCrawler (Figure A.3) is a visualization tool that supports different views on a model, combining metrics and graphs [DEME 99] [LANZ 99]. The tool visualizes entities with shape and color according to metric values combined with different graph layouts. It enables a user to gain insights in large systems in a short time. Furthermore the graphs help to quickly identify source code entities with special combinations of metric values.

A.3 MooseExplorer

MooseExplorer (Figure A.4) provides a uniform way to represent model information [DUCA 00a]. It addresses the problems of navigating large amounts of closely related information. *MooseExplorer* allows a user to browse different entity types in a consistent way. *MooseExplorer* shows for each entity its properties and related entities. A user can click through the entities and thereby further explore related entities.

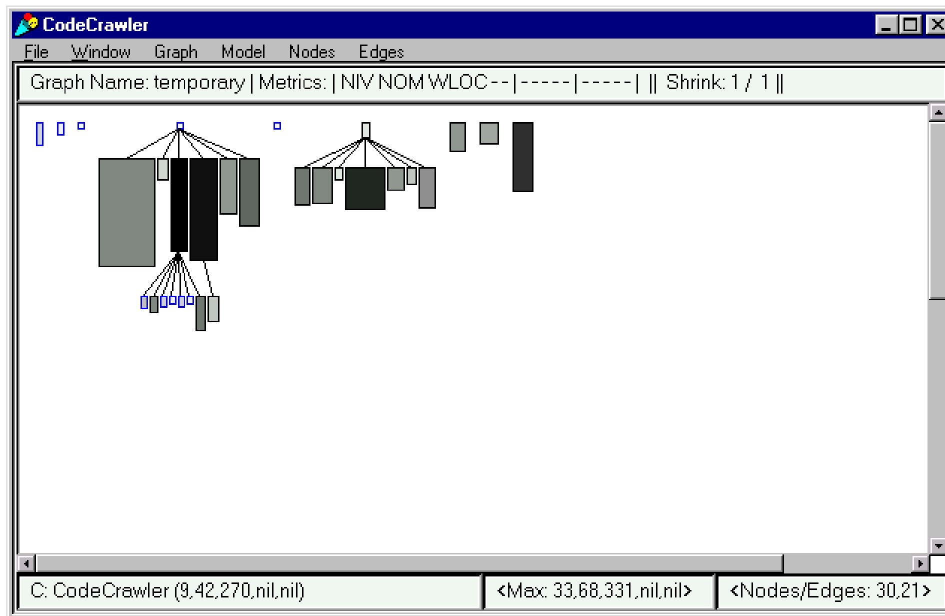


Figure A.3: Screen Capture of *CodeCrawler*.

A.4 MooseFinder

MooseFinder is a query tool that helps to compose queries to retrieve source code entities matching special criteria [LANZ 01b]. Such queries can also be defined on multiple models defining certain change criteria. This tool helped us to gain the evolution facts presented in this work [STEI 01] [DUCA 00b].

The main window of *MooseFinder* (Figure A.5) contains a list with the currently loaded queries. These queries can be applied on a set of entities by pressing the *run* button. The query is applied on entities of the default model defined in the query. The query list contains a basic description of each query. Below in a text field a more detailed description of a query is shown if for the query that is selected. A popup window in the query list offers several manipulations on the query list and selected queries. Every query returns a collection of source code entities. We can choose to which tool this collection is passed to show the output. The default tool is *MooseExplorer*. Alternatively the output collection of entities can be passed to any other application. The output collection can also be passed to the Smalltalk inspector to analyze and manipulate the actual instances of the resulting entities. Another option allows the user to pass output entities directly to the visualization tool CodeCrawler.

The composer user interface (Figure A.6) helps a user to create new queries and to compose complex queries using the queries defined in the list. The query com-

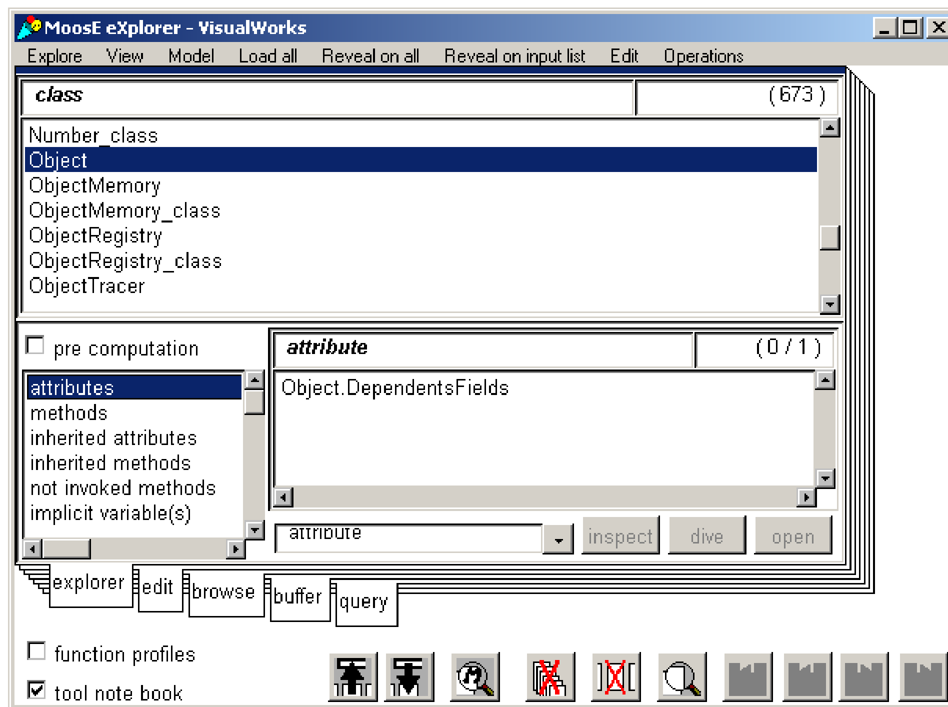


Figure A.4: Screen Capture of *MooseExplorer*.

position window consists of several subpanels, each one covers the configuration of a special type of query.

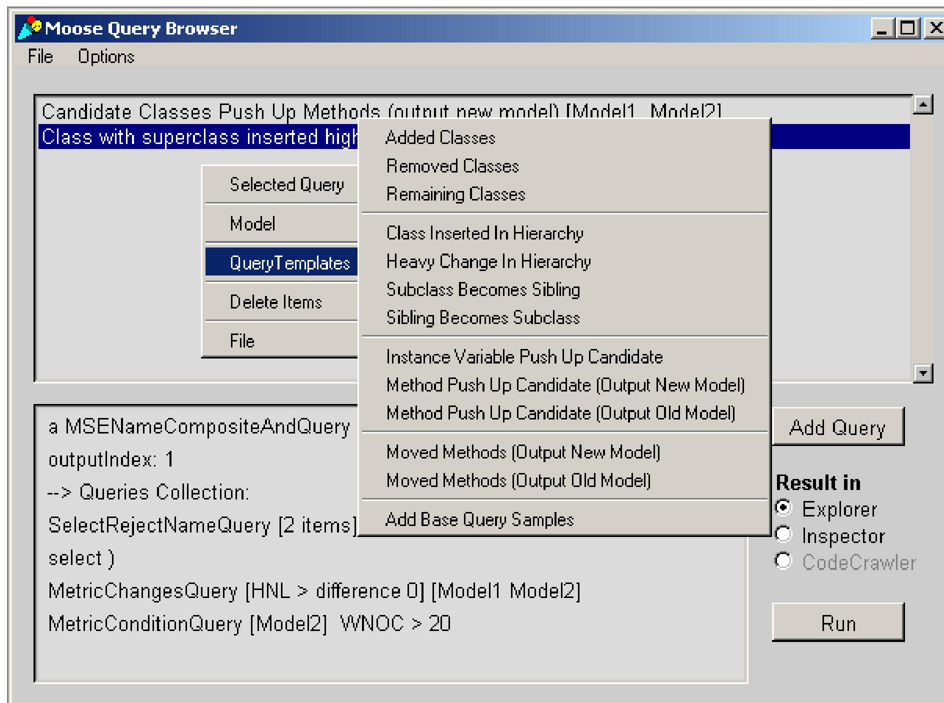


Figure A.5: Screen Capture of *MooseFinder*.

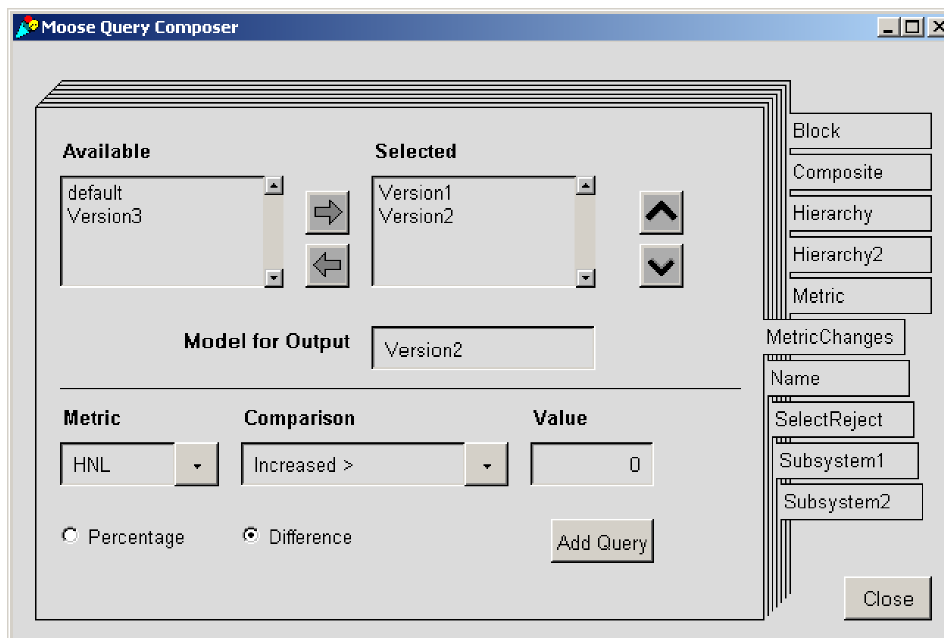


Figure A.6: Screen Capture of *MooseFinder* Query Composer.

Appendix B

SORTIE Report

B.1 Project Background

SORTIE is an established research tool for modeling forest succession. It is actively used in British Columbia, Canada and in the northeastern U.S. to manage forests. The program consists of approximately 28 KLOC of C++ code with sparse documentation. The system has evolved over a long period of time leading to a brittle and complex architecture. Under the lead of the University of Victoria, during a period of six months, developers of state-of-the-art tools from research and industry (Table B.1) were supposed to collaborate closely to analyze the SORTIE system. The first task was to recover the existing architecture of SORTIE. Then, teams were asked to propose a new architecture that is better suited to meeting requirements for future changes.

The background for this collaborative demonstration was based on these main motivating factors:

- Evidence to Encourage Tool Adoption
- Exploration of Tool Interactions
- Promoting Tool Interoperability

From these motivating factors the following six concrete goals were derived:

1. Improve tools and develop better ones by comparing and evaluating existing tools.
2. Provide an opportunity for collaboration and community-building.
3. Learn more about the complementary nature of various tools.
4. Acquire experience with tool interoperability necessary to design an infrastructure for community-wide sharing of tools.

Tool	*Group*	*Institution*	*Contact*
Rigi tool	Rigi group	University of Victoria, Canada	Holger Kienle
cppX	SWAG (Software Architecture Group)	University of Waterloo, Canada	Andrew Malton
TkSee tool	KBRE Group	University of Ottawa, Canada	Sergey Marchenko
Bauhaus tool	Bauhaus Project	University of Stuttgart, Germany	Rainer Koschke
SCG P.U.R.E.	Software Composition Group	University of Berne, Switzerland	Michele Lanza
COLUMBUS/CAN tool	Research group on Artificial Intelligence	Hungarian Academy of Sciences, University of Szeged	Rudolf Ferenc
KLOCwork Suite	KLOCwork group	KLOCwork Solutions Corporation, Canada / US	Nikolai Mansurov
VIBRO (VIsualisation BROker Framework)	Visualisation Research Group	University of Durham, UK	Claire Knight
PBS	SWAG group	University of Waterloo, Canada	Davor Svetinovic

Table B.1: SORTIE Participating Tool Teams.

5. Improve tool evaluation techniques.

6. Encourage the use of the Graph eXchange Language (GXL) [Wint 01].

The results of the tools demonstration were presented at WCRE'2001 held in Stuttgart Germany, October 2-5, 2001. You can find our original submitted report at the end of this appendix. The full reports - also of the other groups - can be found at the project's web site:

http://www.csr.uvic.ca/chisel/collab/

B.2 Project Success

For estimating the overall success of the project we repeat the project's goals. Here they are discussed separately:

1. Improve tools and develop better ones by comparing and evaluating existing tools. Yes we did improve our tools. Comparing and evaluating other tools was rather restricted in seeing other group's results, which mostly could not directly be compared due to quite different original approaches.

2. Provide an opportunity for collaboration and community-building. Like mentioned above. We did another experience in teamwork within our group, separate from that we did not much build a community.

3. Learn more about the complementary nature of various tools. This is interesting. While some tools are source code level oriented, other provide more general graphical system overviews. Some tools focus on browsing, navigation and interaction (SCG), others tend to produce batch processed static views using existing graph drawing frameworks (TkSee). See also Appendix C.

4. Acquire experience with tool interoperability necessary to design an infrastructure for community-wide sharing of tools. Seems to have failed mostly. To our best knowledge not even a GXL file could be shared among the groups. On one hand this seems to come from the fact that the various groups do not extract the same information of a subject system, and on the other hand not every group is ready to deal with GXL. Finally we also had the impression that there was an invisible competition between the groups.

5. Improve tool evaluation techniques. We did not, we just had a quick look at the results of the other groups, but we did not do in a controlled way of first defining criteria and then systematically measure the tools in compliance.

6. Encourage the use of GXL. Maybe, in any case the study shows that in fact if everybody would have been working with GXL, much more work could have been done, if the collaboration would have been one.

Our tools worked fine, parsing was easy, not having the base classes in the model caused no problems for us. After a short time we already had some nice illustrations and believed to have understood the system on an overview level.

There are no bad examples. If we would know an example covering a situation it would not be an example but the situation itself. In the same sense we can learn from every case study. This one gave us some ideas on how to improve the tools, especially it reminded us to some problems concerning models of systems written in C++. Unfortunately the subject system was rather small and makes little use of object-orientated paradigms. Thus many potential aspects of navigating in object-oriented systems could not be seen, however we could learn a lot about other daily problems of a reverse engineer in practice.

Comparison of tools. As a comparison of a couple of state-of-the-art reverse engineering tools the case study gives a good overview of advantages and disadvantages of the different approaches in certain situations.

Community building. A real collaboration where you share knowledge, expertise and experience and build a community failed in our point of view. The reasons are unclear but seem to lie in geographical separation and some sort of competition-like atmosphere among the groups. We can not say if any success was made to-

wards GXL and adapting the tools to it, for we are not willing to do it with our tools for the moment.

Reverse engineering success. We consider the experience for ourselves as successful, we saw many teams with problems in particular with parsing and performant layouts.

Reengineering success. We suggest a new implementation of the system - from that point of view we must say we are not able to propose major restructurings or clever changes to the system.

B.3 SCG Report

(In the following you find the original SORTIE report as it was submitted by the SCG.)

Software Composition Group

SORTIE Report

Michele Lanza, Gabriela Arevalo, Daniel Schweizer, Daniele Talerico

---

Table of Contents

1. Introduction
   1. Description of Tools
   2. Reverse Engineering Team
2. Preliminary Work
3. Analysis
   1. Metric Analysis
   2. Statistical Analysis
4. Sortie Explained
   1. Our View
   2. Conclusion
5. Suggestions & Opinions
6. Conclusion

1. Introduction

1.1. Description of Tools

For our analysis we used the SCG P.U.R.E. toolset written by members and students of the Software Composition Group, as well as one commercial parser.

1. **Sniff+**, a commercial parser and integrated development environment for various languages with which we have parsed the Sortie system.
2. **Moose**, our language independent reengineering environment, written by various members of the SCG since 1998.
3. **CodeCrawler**, a visualization tool which combines visualization techniques with metrics. CodeCrawler is written by Michele Lanza and is based on Moose.
4. **MooseClassifier**, an extension to CodeCrawler written by Daniele Talerico.
5. **MooseExplorer**, a tool which enables textual navigation of Moose Models, developed as part of the diploma by Pietro Malorgio.
6. **MooseFinder**, a query engine which enables us to compose complex queries. It was part of the diploma thesis of Lukas Steiger.
7. **MooseNavigator**, an extension to CodeCrawler written by Daniel Schweizer as part of his diploma thesis.

1.2. Reverse Engineering Team

The team for the Sortie experience was composed of the following people:

1. **Gabriela Arevalo**, did her master on software architecture and is currently doing her Ph.D. on components.
2. **Michele Lanza**, did his master on reverse engineering and is currently doing his Ph.D. on reverse engineering and software evolution.

3. **Daniel Schweizer**, currently doing his diploma thesis on reverse engineering and navigation of metamodels.
4. **Daniele Talerico**, currently doing his diploma thesis on reverse engineering.

2. Preliminary Work

Parsing & Loading

The first step consisted of parsing the source code using Sniff+. Parsing did not pose a major problem. Using a tool called Sniff2Famix we used the symbol table generated by Sniff+ to generate a CDIF file, which contains a textual representation of all software entities contained in Sortie. Using that file we can load Sortie into Moose, our Reengineering Environment. Moose is language-independent. This whole process took less than one hour.

3. Analysis

The analysis made is based on two aspects of the SORTIE system:

1. Structure of the different parts of the system (definition of classes, attributes and methods).
2. Communication and collaboration between different parts of the system.

3.1. Metric Analysis

We ran our metric engine on Sortie, which took a few seconds. An overview of Sortie can be seen in the table below.

Entities	Number
Classes + (Structs)	63 + (6)
Methods	763
Attributes	1935
Functions	5
Inheritance Definitions	10
Invocations	683
Attribute Accesses	6736
Formal Parameters	985
Global Variables	72

Considering the huge number of attributes and attribute accesses and the low number of inheritance relationships, we first made a general analysis of the system, and afterwards a study of specific parts of the system. In the figure below we see a first visualization of the system using CodeCrawler.

From this picture we can see aspects like inheritance, size of the classes (number of defined attributes and methods) and namespaces.
Inheritance defintions: The figure shows the inheritance hierarchy of the system, which is very flat. There are only 10 inheritance definitions.
Namespaces: The colors represent the different (artificial) name spaces we have detected. It seems the classes can be categorized into the following groups:

1. **Dialog Classes** (12) which contain the substrings *Dialog* (10) or *Dlg* (2).
2. **Form Classes** (33) which contain the substrings *Form* (22) or *Fm* (11).
3. **Sortie Classes** (6) which contain the substring *Sortie*
4. **Structs** (6) which have lowercase names with two exceptions: *TGridSubstrate* and *TPlotPoint*
5. **The Rest** (11) which does not fit the above conventions but which in some cases is a plain case of name policy breach, i.e., the classes should have one of the above substrings in their name but they do not.

Size of Classes: In the above view we display all classes of the system and use metrics to render the size of the nodes as follows:

- The wider a class is the more attributes it defines (NOA)
- The taller the class is the more methods it defines (NOM)

Below we see a summary of some metrics of the largest classes of Sortie.

Class Name	*NOA Number of Attributes*	*NOM Number of Methods*	*WLOC Total Lines of Code*	*Average LOC per Method*
TMainWindow	237	78	2099	27
THarvestDialog	124	66	**2746**	42
TSpeciesDialog	**255**	**12**	567	47
TPlotDialog	144	25	744	30

TSortieIO	146	28	**2946**	**105**

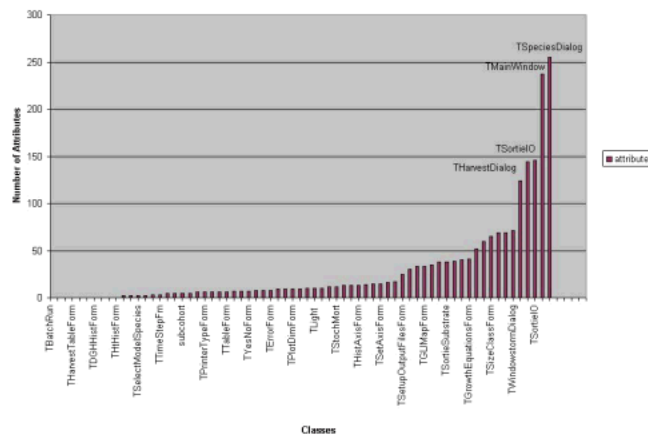
3.1. Statistical Analysis

Focusing more in a deep analysis, and using the number of attributes and methods per class, we proposed to make an analysis of how the class distribution is, seen in an statistical way. In the figures below, we show where the classes (presented in the table) are located in the distribution. Subsequently, we present the following distributions:

1. Number of Defined Attributes per Class
2. Number of Defined Methods per Class
3. Number of Defined Attributes compared to Defined Methods
4. Defined and Invoked Methods

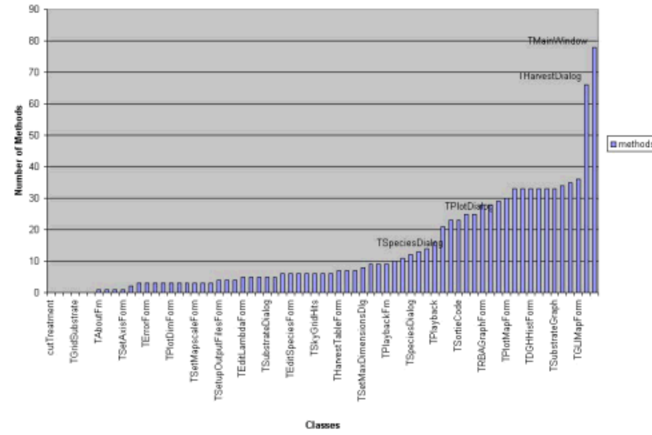
Number of Defined Attributes per Class

Based on the number of attributes defined in each class, the next figure shows how the distribution of the classes is: not uniform. Most of the classes have less than 20 attributes but then we see a high increase. As we saw in the previous table, *TSortieIO* and *TPlotDialog* are two classes with an average of 150 attributes and the classes *THarvestDialog* and *TSpeciesDialog* have approximately 240 and 260 attributes respectively. The largest classes in number of attributes belong to the *Dialog* namespace.



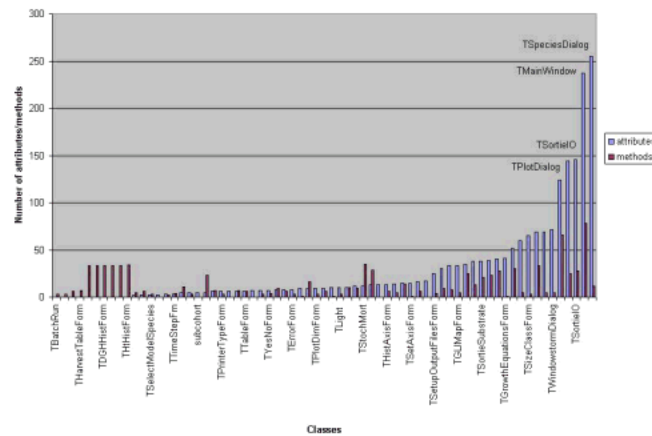
Number of Defined Methods per Class

Thinking in terms of behavior, we analyzed the number of methods defined in the classes. This distribution is more uniform than the previous one, except for the classes *THarvestDialog* and *TMainWindow* that contain approximately 70 and 80 methods respectively. This distribution has the same features as we detected with attributes. The largest classes belong to the *Dialog* namespace.



Number of defined Attributes compared to defined Methods

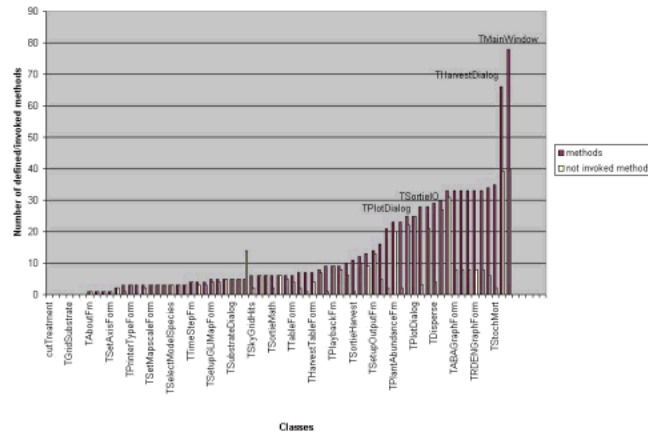
As we saw in the first figure, we saw that if we analyze the number of attributes and methods, many classes seem to be data-containers without almost no behavior. In the first figure we saw that the classes are wider than tall. In the following picture, we present the information in a different way. We see the number of attributes in blue and number of methods in red. For example, the class *TSpeciesDialog* has little behavior compared to the number of defined attributes.



Defined and Invoked Methods

The concept of classes as data-containers can also be seen when we make a comparison between the defined methods and the invoked methods of a class in the system. The next picture shows that only a few of the classes have methods that are invoked in the rest of the system. This fact makes the system appear smaller than it is, if we think in terms of the level of interaction between the classes.

When we see the list of non-invoked methods we see that the most of them are related to the interface communication, for example *OKBtnClick(TObject*)*, *SpeciesListBoxDblClick(TObject*)*, *CancelBtnClick(TObject*)* in the class *TWindstormDialog*. Most of these "non-invoked" methods are called in the files *.dfm, as we verified to get a confirmation.



The Classes *TSetDensitySizeForm, TDisturbanceDialog, TWindowstormDialog, TEditLambdaForm, TTimeStepFm, TPlotDimForm, TSetMapscaleForm, TPrinterTypeForm, TSpatialInterp, TSaveHarvestResultsDialog, TGrowthEquationsForm, TAboutFm and TSavePBFm* have two main features:

1. all their methods are not invoked in the rest of the system
2. their methods do not invoke any method of the rest of the system

As we said previously, these classes seem to belong to the interface part of the system. This reduces the amount of classes that really model the domain of this system.

Global Variables

In the system, we discovered at first 72 global variables. But in a deeper analysis, we see that the number of global variables are close to the number of classes (69). When we have a look at the code, we see declaration like: *extern TTreeMapFm *TreeMapFm* or *TTreeMapFm *TreeMapFm* and *TreeMapFm* is the global variable. The keyword *extern* is used to make local names have external linkage. Like this, in the files classes can be declared local. When they are declared as extern, they can be used outside the scope of the files where they were declared. Thus, in this system, we consider that there are no global variables. We think this is also a sign of inexperience of the developer during the porting of Sortie from C to C++.

Conclusions about the system structure

After looking at the metrics we draw some conclusions:

1. The large number of attributes and the use of GUI classes indicates a mixing of domain model and GUI. This can only be termed as wrong implementation decisions.
2. The large number of lines of code of the classes also shows some present or future problems:

working with large files is bad from a cognitive point of view.
3. The average length of the methods is also somewhat high, in certain cases over 100 (without counting the .h-file). This can be a possible indicator for procedural coding style, or a lack of a refactoring policy by the developer.

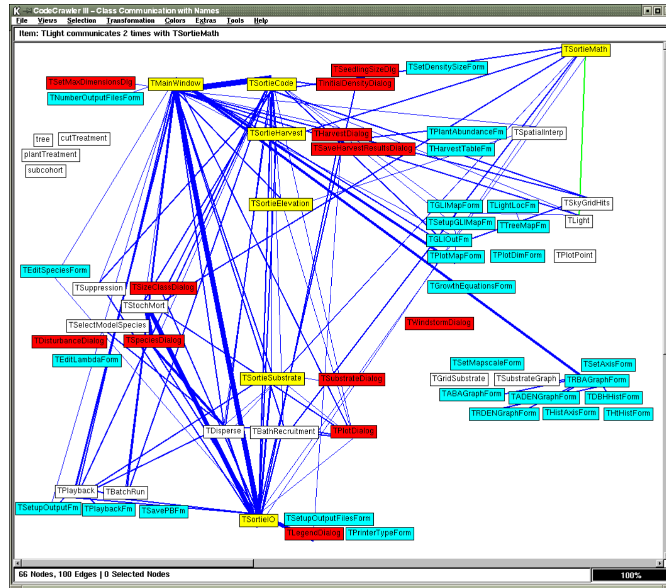
4. Sortie Explained

4.1. Our View

To understand the basic structure of Sortie we looked at the central class called *TMainWindow*, which is also by far the biggest class. This class contains a method called *RunSimulation()*, which is the key to the understanding of Sortie. The whole Program is basically a procedural system written in C++. We know it was ported from C, which strengthens this supposition.
TMainWindow::RunSimulation() contains several calls to certain parts of the system. Although we do not possess any domain knowledge from the comments within this method, it seems like calls to subparts which do this:

1. Harvest
2. Light
3. GLI
4. Bath Light
5. Growth
6. Windstorm
7. Mortality
8. Substrate
9. Disperse
10. Planting
11. Demographics
12. I/O

Using this information we generated the following figure:



In this figure we see all classes and structs of Sortie. The edges represent the invocations between the classes. We obtained this figure after removing 3 classes which do not have any domain, but which get invoked a lot:

1. TYesNoForm
2. TErrorForm
3. TAboutForm

4.2. Conclusion

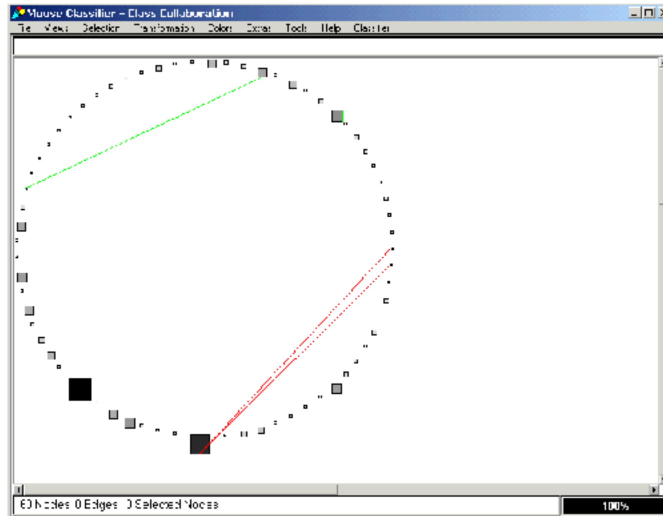
Every subpart of Sortie mentioned above has a similar structure: A class called TSortiexxx (in the figure the yellow ones) which can use Dialog classes (red) and/or Form classes (cyan). With the Dialogs parameters can be set. The Form classes could be there for the Output. The last "piece" is the IO part where files are saved, etc. There's also a "piece" for batch processing. The fact that the program was ported by someone not expert in O-O is indicated by the missing encapsulation (attributes are directly accessed all the time, as good as no private methods), by the flat hierarchies and in general by the non-O-O way of writing code.

5. Suggestions & Opinions

Here is a list of suggestions and opinions about the Sortie system:

1. **Domain mixed with GUI:** We guess that during the port from C to C++ the developer(s) made extensive use of a technology they embraced with too much fervor: The GUI framework of Borland C++. Nearly all classes are subclasses of Borland GUI classes, which results in a dangerous mixing of concerns: Porting the Sortie system to other C++ dialects or even languages will involve problems.
2. **Procedural Coding Style:** In the figure below we see the collaboration relationships between

the classes and structs in the system. What strikes the eye is the low number of edges: 3. This means that the whole system, although written in a (hybrid) object-oriented language does not exploit that paradigm: the functionality within the Sortie classes is not being used by communicating objects, rather by a sequence of classes independent of each other. Noteworthy is also that the collaborations are between classes and structs. Again a sign for procedural thinking.



3. **Domain Dispersion:** A general impression of the system is that the actual domain is dispersed throughout the system. Therefore it is hard to locate a certain aspect of the domain within a certain class or number of classes. This has two negative effects on the system:
   1. **Low extensibility:** If the domain needs to be extended, for example a new type of forest, the developer needs to patch his code in several places.
   2. **Low migration potential:** The dependence introduced by the domain dispersion and further supported by the GUI-guided development makes it nearly to impossible to migrate this product towards another programming language or even C++ dialect.
4. We identified a particularly strange imbalance between data and behavior in a number of classes, some of which we list below:
   - TEditSpeciesForm (16 Attributes, 6 Methods, 63 LOC)
   - *TGrowthEquationsForm (41 Attributes, 1 Methods, 0 LOC)*
   - TSavePBFm (9 Attributes, 1 Methods, 0 LOC)
   - TSetAxisForm (15 Attributes, 1 Methods, 0 LOC)
   - TSetDensitySizeForm (9 Attributes, 6 Methods, 32 LOC)
   - TSizeClassDialog (65 Attributes, 3 Methods, 90 LOC)
   - TWindstormDialog (71 Attributes, 5 Methods, 60 LOC)

   One would expect these classes to be abstract because they introduce methods without implementing them, and contain a lot of data. We can come up with possible reasons for this:
   1. The classes in question are unfinished and still under development.
   2. The GUI-guided development style leads the developer to first define the GUI and then to assign functionalities to the system dependent on the GUI. This can cause holes in the assigned functionalities.
   3. The development style which Borland Builder supports enables the developer to link

GUI elements and functionality using .dfm files. This has the negative effect that the domain is further dispersed within non-source files.

5. The lack of polymorphism is a negative sign: one would expect that polymorphism would be used to model the different kinds of forests, soils, etc., but this seems not to be the case.

6. Conclusion

After looking at the Sortie case study we must say that the reengineering requirements are somewhat unrealistic. We do not think that the code of Sortie can easily be reused, and considered the small size of Sortie would rather propose to rewrite the system using the existing knowledge. For a new architecture we propose to first get a clean notion and implementation of the domain models present in this system and document it thoroughly. This would at least enable to implement new types of forests, etc. with little programming effort. The requirement that non-programmers be able to introduce new types of forests, etc. is unrealistic in this setting and would require a major implementational as well as economical effort to move Sortie towards a framework architecture.

Software Composition Group, 18-09-2001

Appendix C

State-of-the-Art Tools

"No computer can summarize what you tell it."

-Lofti Zadeh, AI-pioneer

C.1 Introduction

In this appendix we present, in alphabetical order, nine state-of-the-art reverse engineering and other navigation tools which we have evaluated. We present the tools in a structured way following a certain template Section C.1.3). You find an overview of all the considered tools, and how they fulfill the requirements in Section C.3. There are many other tools that could have been discussed here. Nevertheless, at the end of the appendix, in Section C.2.8, we present some more noteworthy features of additional tools, without discussing these tools in detail.

C.1.1 Selection

There were four criteria for selecting or rejecting tools to be considered. First, we categorized all the candidate tools. To have a adequate overview of existing tools, we set value on having a broad variation over the categories. Second, we wanted to have the "established market leaders", as a reference for the others. Third, we wanted to check whether there are recent tools with new and innovative features. Fourth, we wanted to experience the look and feel of the tools by ourselves without having to buy any licenses - this is the reason we only discuss tools which are available for free, at least for evaluation purpose. In the following there is an overview of our tool categorization, and why we have chosen which tool for presenting. Each category is supplemented by a table of other tools in the same category. Some of these tools were also partly evaluated, from some we just know that they exist. The sets are a subjective selection among many other possible tools.

- **Knowledge Management:** In the category of information and knowledge management tools, there is a bunch of commercial enterprise tools, which

we did not consider. Then there are a couple of navigators for topic maps, from which we have chosen one of the most traditional, but also innovative products: **The Brain** is a lightweight knowledge management tool. Its model independent architecture is open to be adapted for many purposes.

Application	*Producer*
Brainware	SER
FreeMind	Joerg Mueller
grapeVINE	grapeVINE
Inxight	Inxight
K2	Verity
K42	Empolis
KnowledgeMiner	USU
KnownSpace	KnownSpace Group
Omnigator	Ontopia
SemanText	Eric Freese
The Brain	TheBrain Technologies
Topic Map Designer	Ontopia
Topic Navigator	Mondeca
TM4J	Ontopia

Table C.1: Tools of the Category Knowledge Management.

- **Text Browsing:** One of the most traditional navigation tools is Netscape Navigator. It was built to explore the World Wide Web (WWW), based on text and hyperlinks. In combination with **Javadoc** it represents a very simple, but depending on the current task, still quite powerful reverse engineering tool. The search engine Google provides a useful browser add-on. This additional toolbar, unfortunately, is yet only available for Microsoft's **Internet Explorer**.

Application	*Producer*
CXREF	Andrew M. Bishop
Internet Explorer	Microsoft
Javadoc	Sun Microsystems
Netscape Navigator	Netscape
PBS	University of Waterloo, Canada
Xrefactory	Xref-Tech

Table C.2: Tools of the Category Text Browsing.

- **Graph & Meta Model Browsing:** This is the category of most popular reverse engineering tools. It seems to be the classical approach to generate meta models for representing a subject system. All the views, queries and metrics are then operated on this meta model. Usually graphs are used for visualization. **Rigi** is probably the best known reverse engineering tool. Because of that, it is a viable candidate for the comparison with its newer competitors. **SHriMP** is the C++ to Java port of Rigi, and can be seen as

its successor. Since the two tools follow a contrary approach for navigation, we present them both. An instance of the latest commercial competitors in this category, is **Small Worlds** which provides many reverse engineering features.

Application	*Producer*
CIAO	AT&T Labs-Research
CodeRover	MKS
CodeSurfer	GrammaTech
CPPX	University of Waterloo, Canada
Datrix	Bell Canada
daVinci	University of Bremen, Germany
GEN++	University of California, USA
GOOSE	University of Karlsruhe, Germany
GraphTool	University of Durham, United Kingdom
Headway	Headway Software
Imagix	Imagix Corporation
inSight	University of Moscow, Russia
Rigi	University of Victoria, Canada
SHriMP	University of Victoria, Canada
Small Worlds	Information Laboratory
Source Browser	Swiss Federal Institute of Technology Zurich
Source Code Browser	Aubjex
Source Explorer	Intland
Source Insight	Source Dynamics
Source-Navigator	Red Hat
SPOOL	University of Montreal, Canada
TkSee	University of Ottawa, Canada
Visualize it!	Power software

Table C.3: Tools of the Category Source Code Browsing.

- **Integrated Development Environment (IDE):** IDEs have left the stage of being simple text editors with syntax highlighting. Today's IDEs cover a wide range of useful features to support the software development process. Although most of them do not provide any graphical representation of a system, the features for navigation are often quite remarkable. We have chosen to present **Eclipse** because it is a recent open source project, and we believe that many of us will use it as the extensible standard IDE of the next years, to develop Java and other programming languages. Many years of experience in the development of the VisualAge IDE palette find their application in Eclipse.

- **UML Modeling:** The Unified Modeling Language (UML) is the standard graphical notation for object-oriented design. Several tools support graphical modeling and development. Probably one of the most evolved tools in this set, is **TogetherJ**, which offers full round trip engineering, and for that was selected by us for presentation.

Application	*Producer*
Delphi	Borland
Eclipse	Eclipse Project
Juliet	Infotectonica
VisualWorks	Cincom
SNiFF+	Wind River Systems
StP	Aonix
Understand C & Co.	Scientific Toolworks
Visual Studio.NET	Microsoft

Table C.4: Tools of the Category IDE.

Application	*Producer*
ArgoUML	Tigris
Rose	Rational
Together	TogetherSoft
XDE	Rational

Table C.5: Tools of the Category UML Modeling.

- **Static Analysis / Documentation:** Finally there are tools for source code analysis or documentation. Their quality is in creating static result sets, lists, and other representations. We decided to present none of this group, since these tools usually do not offer any interactive navigation possibilities.

Application	*Producer*
Insure+ & Co.	ParaSoft
KLOCwork	KLOCwork
Logiscope	Telelogic
PL/I Analyzer	Phoenix Software Technologists
Refactorit	Aqris
Tarantula	Georgia Institute of Technology, USA

Table C.6: Tools of the Category Static Analysis / Documentation.

C.1.2 Scope

Our focus lies on navigation features. What we do not consider in this work, are issues of performance or enhanced reverse engineering features, like hotspot search, pattern discovery, impact, or dependency analysis. A state-of-the-art in reverse engineering, which also covers such issues, is provided by Si-Triet Nguyen [NGUY 00]. Nguyen, focuses on ways to parse- , model- , and present big amounts of information, and thereby, partly treats the same tools like we do.

C.1.3 Template

Each tool is presented in the following structured template:

Application: The name and a short description introduce each tool, and point to its originators, and relevant publications.

Screen capture: A screen capture gives an impression on how the application looks like.

Description: We then describe the tool in more details. We identify the tool's family, its main purpose, the project or product background, and platform availability. Each tool's functionality is considered according parsing, modeling, and representation of information. The description is completed by a pointer to the project's or product's web site, and other resources.

Relations: After that, we present related tools, involved technology, and extensions, adaptations, add-ons, or plug-ins, if there are. This is also the place for success stories.

Discussion: Finally, we measure the tool against the concerns of building navigation tools for reverse engineering (Table 2.5). We list the best, the worst, and the most noticeable characteristics and features.

C.2 Tools

C.2.1 Eclipse

Application: The Eclipse open source platform provides building blocks and a foundation for constructing and running integrated software-development tools. The Eclipse/JDT project provides the tool plug-ins that implement a Java IDE [OTI 01].

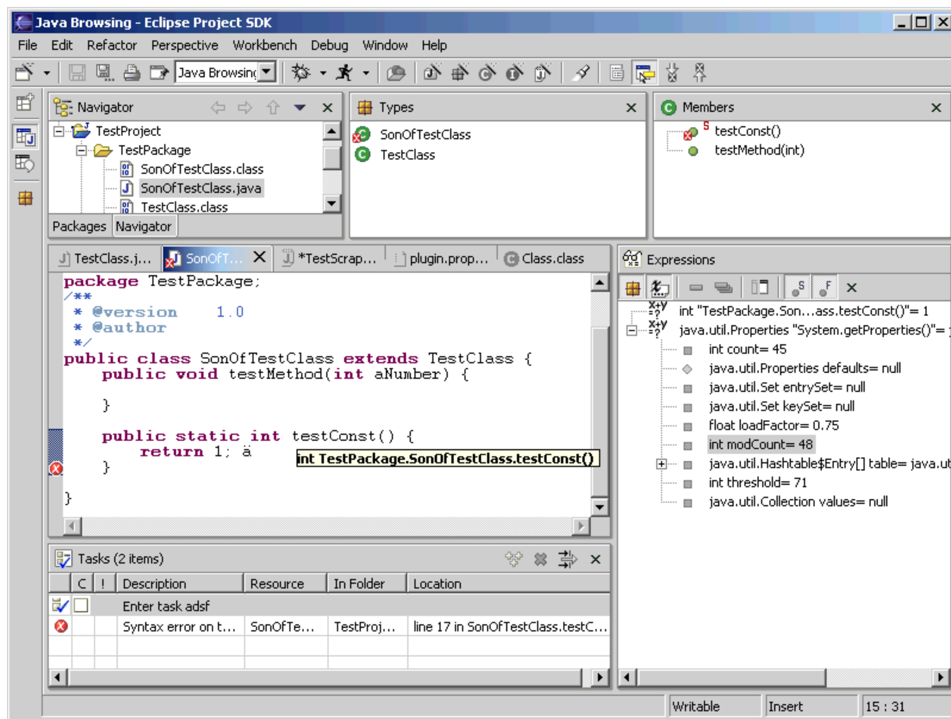


Figure C.1: Screen Capture of Eclipse.

Description: The Eclipse Platform is designed for building IDEs that can be used to create diverse applications as web sites, embedded Java programs, C++ programs, and Enterprise JavaBeans. A plug-in is the smallest unit of Eclipse Platform function that can be developed and delivered separately. Usually a small tool is written as a single plug-in, whereas a complex tool has its functionality split across several plug-ins. Except for a small kernel known as the Platform Runtime, all of the Eclipse Platform's functionality is located in plug-ins.

http://www.eclipse.org/

Relations: The Java development tool Eclipse/JDT, is what IBM delivers as *WebSphere Studio Application Developer* which can be seen as the successor product for both *VisualAgeJava* and *WebSphere Studio*.

Low Entry Barriers: Eclipse is free. This might be a first reason why you download it and try it out. It is open source and thus independent from certain manufacturers or vendors. Download and installation worked without any problems, from scratch. The trust increases after launching a new program, when a short tutorial guides you through the most important features of the tool, and successfully ends in a sample project. Wizards help to perform tasks for the first time. A concise and complete documentation in form of an integrated online help, gives you the certainty of being able to solve your problems, also if the solution is not obvious at the moment. When in the future more useful reverse engineering features will be provided, as plug-ins, they will probably also be used, since they only ease a developer's daily work, and make it not more complicated.

Completeness: The model of Eclipse is the total of source files and folders, thus the model is complete. Basically Eclipse is a set of editors that are opened - , and a set of functions that are performed directly on these files. No meta model is created. The number of features is adequate, and can be extended.

Simplicity: The user interface is nice and intuitive. Menus and commands are like we are used to from other editors or tools. "Saving" a class does not only save the source file but, in the background, without that the user has to be aware of that, automatically compiles the code and generates lists of warnings, or errors, and updates the parse tree.

Navigation between Tool States: Eclipse keeps an account of all visited views (in Eclipse views are the subparts of the window), and editor (in Eclipse editors are instances of views, one for each file). Navigation is possible to the next, the previous, or to a specific element from the history list. Undo and redo of actions is possible.

Navigation in Graphs: Eclipse currently supports no graphical views on its content. The only sight are tree views. Collapsing or out folding of branches is possible.

Navigation in Object-Oriented Models: Several prepared views show code artifacts in hierarchical trees. Navigation along *declarations* and *references* for all artifacts is possible. Additional navigation to super class, super method, and Javadoc is supported.

Efficiency: Hints explain the tool bar buttons. Popup combo boxes help you for the auto completion while typing complicated class names or signatures of methods. Refactoring support is not only a question of saving time, but also for reducing the chance to make oversights. Testing the preconditions, and if everything is ok, to perform the task automatically is a complicated task that for humans, but if well implemented, easy for a computer.

Feedback: In contrast to the other tools discussed here, Eclipse has its primary purpose in forward engineering. This involves that the user can modify the system under consideration, and feed back his experience. Four additional ways of feedback are provided. First, bookmarks can be set, to refind previous points of interest. Second, annotations can be written directly into the source code. Third, Javadocs can be created for each class. Fourth, tasks can be added, to not forget about things to do.

Classification: Various configurable perspectives give different views on a system, depending on the current task at hand. The task list can be filtered by type, location, resource, problem severity, task priority, and task status.

Complexity Reduction: Source code is organized in packages and projects. The big number of lines of code in a project is split into parts, from which the smallest is the method. These artifacts can be displayed in various tree views. Several perspectives are predefined, like *"Java Browsing"*, *"Resource"*, *"Team"*, or *"Help"*, additional perspectives can be configured. Syntax highlighting eases the readability of the code, and contributes to recognizing typing errors sooner. Wizards and assistants help to perform rarely used tasks, like setting up a new scrap book page. Scrap book pages are a concept for running and inspecting Java expressions under the control of the debugger, similar to a Smalltalk workspace, although only performing static methods.

Consistency: A project can be closed and later reopened. Projects can be transferred, since everything is stored in configuration files. A weak form of geographical consistency is provided in the trees, since all the containers and code artifacts are always sorted alphabetically. A local history with the granularity on method level, lets you reload previous version of source, if you want to discard your recent work, or just compare versions and see the recent modifications. As repository for developer teams, Eclipse currently supports the Concurrent Versions System (CVS), where you lift the granularity to class level at the time of releasing your

work, since CVS is file-based, and Eclipse splits the Java class files only for internal processing, and not physically.

Memory: Actions can be redone, or undone. A history of visited editors and views, as wells as lists of breakpoints, bookmarks, tasks, problems, errors, or warnings is kept. Further supported navigation is back and forward in the history of visited code artifacts in the tree view, as well as jumping to an artifact's container.

Storage: Eclipse stores everything in regular files and folders. The workspace configuration is stored on the local machine, the source may be a mix of local files and divers CVS repositories on various remote servers.

Extensibility: Eclipse is designed to be a platform for creating your own extensible IDE. We distinguish between configuration and extension. Configurable toolbars, keyboard-shortcuts, or perspectives build a first category of possibilities to adapt the tool to your specific needs. Much more flexibility is provided by the fact that Eclipse is open source. You can create your own, plug-ins, like implementing a new refactoring; new tools, like the support for an additional programming language; or simply do small modifications in features and behavior. This is the first Java IDE that in terms of extensibility comes, close to what people are used from Smalltalk development environments

C.2.2 Javadoc

Application: Javadoc is the tool from Sun Microsystems for generating API documentation in HTML format from doc comments in source code. Internet Explorer is a free web browser from Microsoft. The Google Toolbar increases your ability to find information.

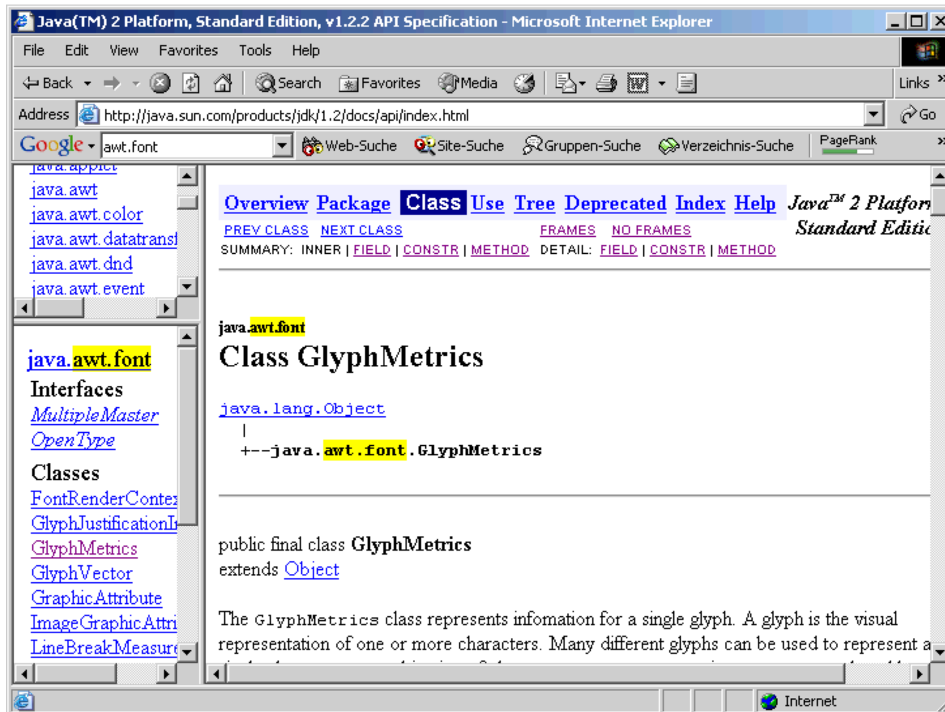


Figure C.2: Screen Capture of Internet Explorer with Google Toolbar & Javadoc.

Description: Today, Microsoft's Internet Explorer has by far the greatest share of web browsers in use. This is the reason why useful add-ons like the Google Toolbar, or its counterpart, the Alexa Toolbar[1], are not yet available i.e., for Netscape Navigator. After all, Google announced to port its toolbar to other platforms and browsers. Among other things, Internet Explorer provides a wide range of features for the navigation in a world of texts and hyperlinks.

`http://www.microsoft.com/ie/`

[1] `http://info.alexa.com/`

The Google information portal integrates an Internet search engine, active & archived discussion groups, and a web directory. With the Google Toolbar, all these services are accessible directly, from within the web browser. Enhanced features bring real added value, among them are a page ranking, additional page information like similar pages, pages that link back to that page, or highlighting and finding occurrences of search terms on the page.

http://toolbar.google.com/

Javadoc is a tool that parses the declarations and documentation comments in a set of source files, and produces a set of pages in the Hypertext Markup Language (HTML), describing the classes, inner classes, interfaces, constructors, methods, and fields - each occurrence of a source code artifact is represented by a hyperlink to the artifact's own description.

http://java.sun.com/j2se/javadoc/

Relations: Another lightweight approach for navigation in C and Java source code by adapting existing tools, is *Xrefactory*[2], which is in fact a refactoring browser for Emacs, with the ability to generate Javadocs and other HTML reports. Limited to cross-references from C program source code, an alternative program that produces documentation in LaTeX, HTML, RTF (Rich Text Format) or SGML (Standard Generalized Markup Language), is *Cxref*[3].

Low Entry Barriers: Because web browsers belong to the most frequently used navigation tools in the world, they are viable candidates to our considerations. First, no additional installation effort is needed to work with your favorite web browser, on your favorite machine and platform. Second web browsers are highly tested and running quite stable, at least at browsing standard HTML. Third, no extra financial resources are required, since most of the web browsers are free. Many IDEs support the automatic Javadoc extraction. In many Java projects, Javadocs have been adopted. Major software developing companies, like Netscape or Sun Microsystems, refer to reference documentation on their products, in form of Javadocs, accessible over the Internet, where it is easier to keep them up-to-date. With the Google Toolbar installed, you can translate a page to your preferred language. We appreciate that feature, however translators are still not precise enough,

[2] http://www.xref.sk/
[3] http://www.gedanken.demon.co.uk/cxref/

to yield useful results. Browsing a Javadoc might even be comfortable for non-technical people to get an overview of a system, or to look up a certain specification detail. Because of the read-only character, the risk to harm source code, is eliminated.

Completeness: The model is as good as the developers contribute to document and tag their code. After all, the DocCheck Doclet checks doc comments in source files and generates a report listing the errors and irregularities it finds. By defining your own Javadoc Doclets and introducing custom tags, you can theoretically reflect all information available about a system in your Javadocs, including paths for navigation among the pieces of information.

Simplicity: Probably every software developer is familiar with browsing HTML sites, thus browsing a Javadoc should be easy. No complicated representations or unfamiliar tool handling requires the formation of new habit. Once a Javadoc is created, browsing is fast.

Navigation between Tool States: Like at browsing other sites on the Internet, users can browse a Javadoc - creating bookmarks, inspecting the history of visited pages, going back and forward, and opening a link in a new window.

Navigation in Graphs: The web can also be represented in graphs. The pages correspond to nodes, the hyperlinks correspond to edges. The navigation along edges is one of the primary principles of the web. Internet Explorer provides a feature *"Show Related Links"* which asks the Alexa information portal for a list of related links, and providing other background information about a page like the number of referring sites, the date of first indexing, and the date of the last modification. This feature is similar to the "What's Related" feature in Netscape Navigator. The Google Toolbar implements additional comfort, e.g., navigation to the parent directory of a page, listing all indexed pages that have a link on the current page, and toggle from one occurrence of search terms to the next. A word or text can be selected, the context menu provides the possibility to turn that directly over, as search criteria for a new Google search. The context menu above a hyperlink offers navigation to similar pages, to a cached snapshot of the page, or to backward links.

Navigation in Object-Oriented Models: Similar to what we state in the paragraph "Completeness", the richness of supported navigation depends on the way of extracting information out of the system and on the way of generating Javadocs. The default Javadoc parser only regards the most common relations. This first category of navigation steps includes package, class, uses, subclasses, super classes, implementing classes, interface, constructor, fields, inherited fields, field types, methods, return types, and argument types. Another category of navigation steps

is provided by extra tags, that the programmers have to set at development time. These include *"See Also"* sections, in-lined texts, and links to thrown exceptions. Further optional predefined tags include author, version, information regarding the version of the Java Developer Kit (JDK) and deprecation.

Efficiency: We believe that Javadoc is a efficient tool to look up implementation and comments, because it is easy to use and the respond times are very short. The variety of navigation features provided by the web browser in combination with detailed references of Javadocs support efficient searching of relevant information.

Feedback: If you want to find out about the navigation behavior, of you want to additionally annotate what you see in a Javadoc, you must do that externally. Bookmarks can be set at position of interest, however these will not work as wanted if browsing with the convenient frame-based view. The same is true for the built-in page history of the web browser. For linking to further documents or data, you may also prepare tags, that automatically inline external information while regenerating a Javadoc. Javadoc itself is a read-only concept. To integrate new information about the system the model must be changed, and the model is the source code which has to be modified, re-tagged, or re-commented.

Classification: The highlighting of search terms as they appear on the page helps to find the relevant text positions. The page ranking makes a statement about the popularity of a page, in measuring how many other pages are linking to that page, and how their ranking is, and so forth. Unfortunately, the information portals do not exactly specify how they compute similarity among pages. However, all above features for relevance work only if the subject Javadoc was indexed by the information portal that provides the toolbar, which is for example the case for Google and the JDK 1.2 API Documentation[4], but usually not in your individual projects. Based on the Internet Explorer settings for security, privacy, and content, you may specify additional rules for filtering information.

Complexity Reduction: The frame-based view divides the window in three sections. The first is listing the packages, another is listing the classes in the selected package, and the third shows the selected class whatever the users chooses to display in the main frame. Alternatively a system can be displayed in a frame-less page, by a big tree view, or by a complete alphabetical index. Javadoc can run with several parameters for different levels of detail. Once created, it is not possible to hide and un-hide aspects from a Javadoc. For reducing the complexity no source code is shown - only method signatures, interfaces and documentation.

[4] http://java.sun.com/products/jdk/1.2/docs/api/index.html

Consistency: A Javadoc is consistent to the source base as long as nobody modifies the model. To be up to date, a new Javadoc has to be generated after every change. Consistency in handling is given by use of a standard web browser and simple HTML pages. The content is not dynamic and thus will be exactly the same, when revisiting later. The Javadoc can be transferred, it finally consists of one folder containing a couple of files and sub folders.

Memory: Internet Explorer keeps track of visited pages, the history can be sorted by date, size, most visited, or order visited today. The bookmarks, or favorites, like they are called in Microsoft's jargon, let you additionally set pointers to important resources that you want to refined directly. Another list that is kept is the list of search terms.

Storage: A whole Javadoc can be physically stored in an archive file. Javadocs are typically put on a web server, accessible over the Internet, or in a private network. However, Javadocs can also be stored locally to be able to work offline.

Extensibility: The menus and features of Internet Explorer can be extended by installing an additional toolbar, providing more navigation features, and accessing extra information about pages from an index. Javadocs can be adapted and extended primarily by two concepts. First, Doclets serve as layout and format templates, defining how to present which information, similar to Java Server Pages (JSPs). Javadoc provides default Doclets for the creation of HTML, XML, MIF, RTF, as well as FrameMaker and PDF documents. Second, custom tags allow the developers to add new semantics to Javadocs. These custom tags must be considered in the adapted Doclets, for the Javadoc reader can profit.

C.2.3 Rigi

Application: Rigi is a visual tool for understanding legacy systems, maintained by the University of Victoria, Canada [WONG 98].

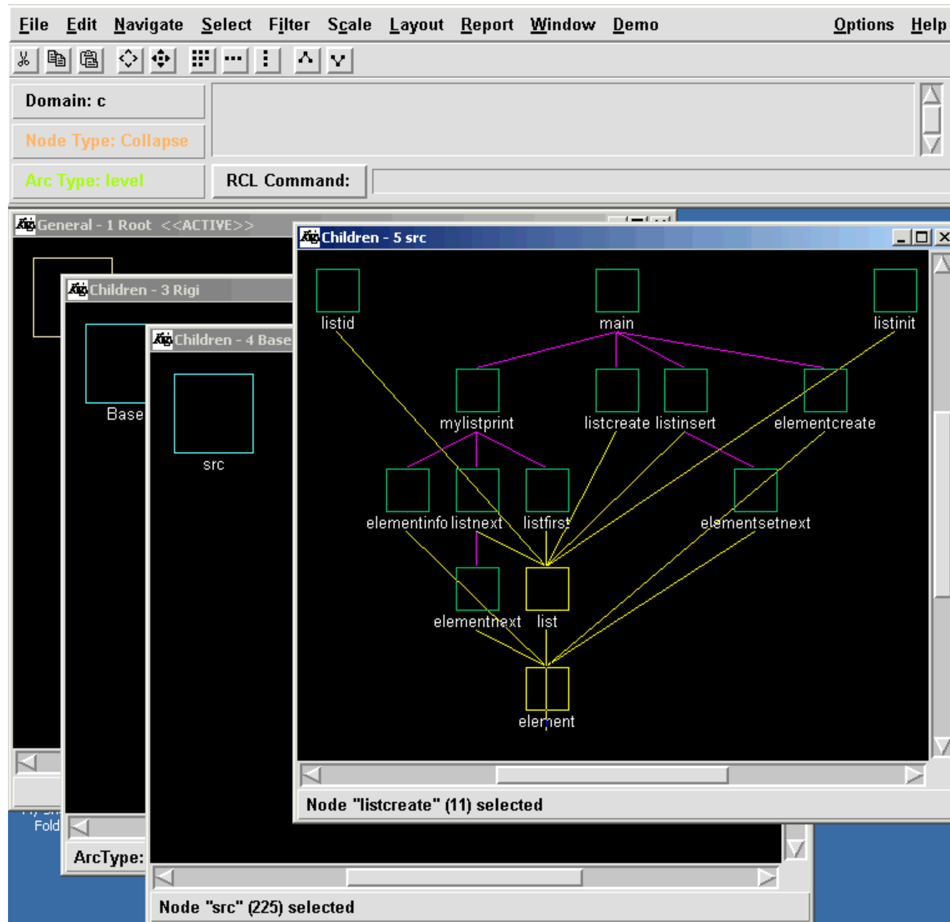


Figure C.3: Screen Capture of Rigi.

Description: Rigi is one of the most traditional reverse engineering platforms, and has its origin in a PHD thesis dating back to 1986 [MÜLL 86]. Rigi displays a system disjointed in graphs of nested boxes, representing containers and artifacts of software. *Rigireverse* is a set of parsers for C, C++, COBOL, and Java. The artifacts are stored in an underlying repository on a *Rigiserver*. *Rigiedit* is the graphical editor which provides editing, manipulation, annotation, hypertext,

and exploration capabilities. Versions of Rigi are available for Sun SPARCstations (SunOS), IBM RISC System 6000 (AIX) workstations, and PC-compatible (Windows 95, Windows NT, Linux 2.x) machines.

http://www.rigi.csc.uvic.ca/

Relations: Rigi software repositories are stored in the Rigi Standard Format (RSF), RSF is also the standard file format for *SHriMP* Section C.2.4 - the C to Java port of Rigi. Graph Exchange Language [WINT 01] (GXL) to RSF converters are promised to be available in the near future[5].

Low Entry Barriers: After installation, clear instructions are provided. Three illustrative demo tours which guide you through the key features and views. We soon feel familiar to use Rigi. However, for being able to illustrate own software projects, more effort is needed. The source has to be parsed to generate the RSF repository. Since operating not directly on the source code, but on a meta model, harming any content is not possible. This might give certain users additional safety.

Completeness: At first sight, we did not miss any information about a system. The number of features is concise, and allows us quickly to display some graphs, apply layouts, and already collapse some nodes to an container.

Simplicity: Operation with windows, graphs and nodes is intuitive, the names of the menus entries seem clear. The concept of nested boxes is simple to understand. Views can be generated showing the dependencies of multiple boxes. To these graphs, various layouts can be applied to all nodes, or the selected subset of nodes.

Navigation between Tool States: Rigi uses the multi-window interface concept [STOR 96]. Each window is showing one single view. Double-clicking on a node, e.g., opens a new window with the detail contents of a clicked node. Diving to a selection of nodes is possible. Navigation between views is done by changing the focus from one window to the other. Each window provides scrolling and zooming. The windows can be cascaded.

Navigation in Graphs: Rigi views consist of boxes (nodes), and directed arcs (edges). Rigi supports navigation along child nodes, parent nodes, neighbor nodes in general, or diving to a selection of artifacts. Selections can be made manually or

[5] http://www.gupro.de/GXL/

automatically. Automatic ways include *Incoming-, Outcoming Nodes, Forward-, Reverse Tree, By Attribute..., By Structure...*, and *By Name....* Selections can be complemented. A selection of nodes can be arranged vertically or horizontally. Beside various simple grids and hierarchical trees, more sophisticated graph layout algorithms are provided, including spring [GIUS 99], and Sugiyama [SUGI 81]. Nodes can be copied and pasted from one to another view.

Navigation in Object-Oriented Models: A Rigi session starts with a window showing a top level box which is representing the system under consideration, double-clicking into it, opens another window showing the classes of the system. Double-clicking again on such a class box, open another window showing the methods and attributes of the corresponding class. The context menu of each node and each arc, provides ways to open a window for viewing additional structural information about the source code artifact, and for editing the attributes, the annotation, or the source (nodes only). Per default, 21 node types are defined, among them *Collapse, System, Release. Procedure*, or *Module*. The 10 predefined arc types are *any, call, data, structure, syntactic, block, include, composite, refractive*, and *level*.

Efficiency: For every view it is possible to view a "Window Statistics" which lists all the nodes and arcs, split by their type. Another predefined report is "Graph Quality" which gives an overview of the systems composition and encapsulation. The "Exact Interface" presents the interface of a box. This is helpful if the box is representing an abstract data type or simply a set of collapsed nodes. For facilitating reuse, expert users can write scripts in Tcl/Tk using the Rigi Command Library (RCL).

Feedback: Users may annotate nodes and arcs, for elements from which the system does not know the type, the type may be assigned. The artifacts' attributes' values can be edited. Also artifacts can be annotated.

Classification: The concept of affinity is applied in spring layout algorithms e.g., by taking *calls* as a measure for computing the coupling between nodes.

Complexity Reduction: The top down approach of Rigi help a user to start with one simple single box representing the system, and then slowly dive into the parts of interest. Interactive view manipulations are helpful for arranging nodes according to the current preference. Layouting selections of nodes further help. Renaming nodes can facilitate to recognize the nodes faster, and in a familiar naming scheme. Nodes are colored according their type. Although this brings additional information in the same view - it can reduce the complexity. Good layout algorithms can have a great impact on the understandability of a view [STOR 95a]. They help us to arrange nodes in a concise way, with respect to their affinity. With

the multi-window interface concept, the concurrently displayed information per window usually keeps manageable. Collapsing nodes and creating components or abstract data types can further reduce the complexity, by modularizing a system. Finally, filtering information according to the type, or hiding a specific selection of elements can ease understanding by reducing the amount of displayed information.

Memory: Source code artifacts can be edited. Its attributes values can be set, and they can be annotated. This information is memorized by the system. If a view is considered to be useful, it can be left in that window, and a new windows can be opened to further process the same graph.

Storage: The complete system is stored in graphs. These graphs are stored in RSF files. Graphs can be loaded and saved. Additionally you can save and load single views. Reports, annotations, and statistics are stored in text files.

Extensibility: The domain model specifies the entity types and relationships of interest. A certain set is predefined, however this means you can define also your own semantics, as long as your semantic is conform to RSF graphs. With the Rigi Command Library (RCL) users can define and automate common operations on graphs. The scripts are be written in Tcl/Tk. In fact such scripts are also the way how Rigi itself controls the underlying repository and how it creates the nice views of the demos. With the RCL, you have access to the complete functionality of Rigi. Even the adaption and creation of new widgets is possible.

C.2.4 SHriMP

Application: SHriMP (Simple Hierarchical Multi-Perspective) is an Interactive and Customizable Environment for Software Exploration, developed at the University of Victoria, Canada [Stor 95b]

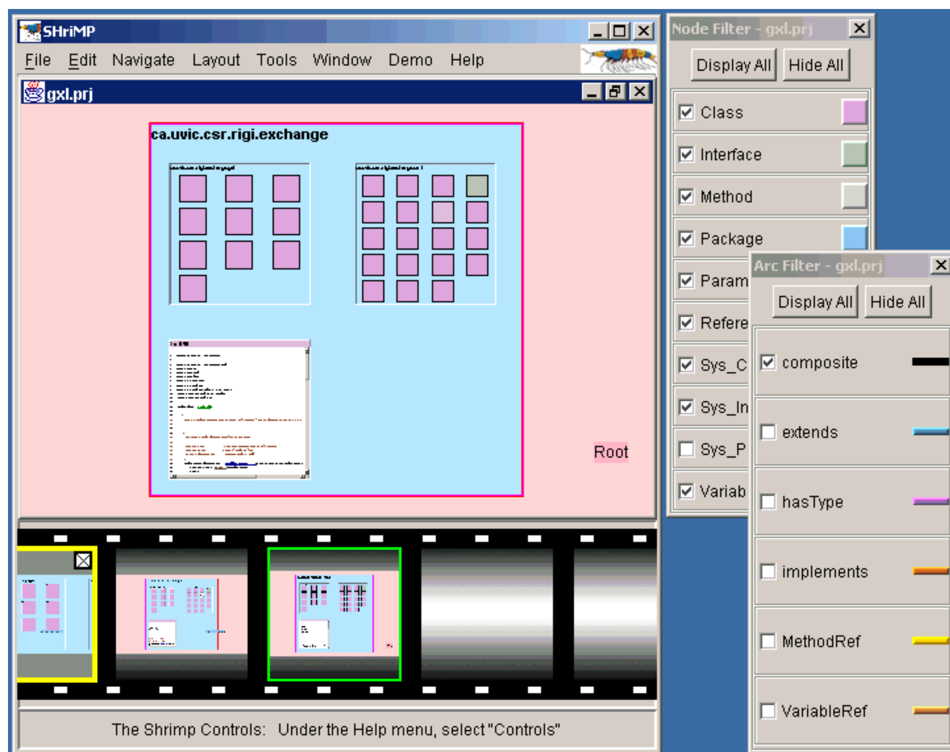


Figure C.4: Screen Capture of SHriMP.

Description: The SHriMP visualization technique was originally designed to enhance how programmers understand programs. The research tool presents a nested graph view of a software architecture. Program source code and documentation are presented by embedding marked up text fragments within the nodes of the nested graph. Finer connections among these fragments are represented by a network that is navigated using a hypertext link-following metaphor. 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. As information model, SHriMP uses the Rigi Standard Format (RSF) which comes from

the reverse engineering tool *Rigi*, described in Section C.2.3. SHriMP is platform independent, and based on Java version 1.3.

http://www.csr.uvic.ca/shrimpviews/

Relations: SHriMP has been integrated with the *IBM Websphere Studio Workbench* which was open-sourced to *Eclipse*, as described in Section C.2.1. In this integration, SHriMP has been customized and re-targeted for visualizing ”flow diagrams”. Flow diagrams are e.g., used in an eBusiness project to model the dynamic aspects of a system, such as the main activities and the movement of information in a business process [RAYS 01].

Jambalaya is the keyword to describe a project to integrate SHriMP with the *Protégé* tool, developed at Stanford University [NOY 00]. *Protégé* is a general-purpose knowledge based tool which allows domain experts to build knowledge-based systems by creating and modifying reusable ontologies and problem-solving methods. This collaboration will result in an environment for acquiring and browsing knowledge pertinent for cancer clinical trials and other applications [NOY 01].

Jazz is a open source graphical framework for Java, developed by the University of Maryland [BEDE 00]. SHriMP currently supports animated panning and zooming, using *Jazz*.

The provided *Java Extractor* relies on combining the results of tools created by third parties. It brings together the data that is extracted by *JavaRE, Javasrc*, and *Javadoc* through the *RigiConvert* tool and generates RSF. *JavaRE - Java Round trip Engineering* is an open source toolkit for round trip engineering by Andersson[6]. *Javasrc - HTML Java Cross Reference* is a tool that parses plain source code to generate HTML'ized source code[7]. *Javadoc* is described in (Section C.2.2).

Low Entry Barriers: To get a first impression on how to use SHriMP, the product web site provides a short introduction, some user scenarios, and even example videos of two exploring sessions. After downloading, the provided installer does his work. At launching the program, the screen gets tiled by three windows, and a fourth popup window explaining the controls. Beside the main window showing the content, the other two windows are for explaining the colors, and to configure the visibility of the types. One window for the node types, the other window for

[6] http://javare.sourceforge.net/index.php

[7] http://home.austin.rr.com/kjohnston/javasrc.htm

the edge types. Help is available in the menu, linking to a brief manual provided on the project's web site. The main menu offers four demo models for exploration. The safety of the content is guaranteed, since work is only done on meta model. For exploring the own projects source code, the extractors must create the models first. A offline manual for the *Java Extractor* is provided.

Completeness: SHriMP uses the Rigi Standard Format (RSF) model and parsers. The *Java Extractor* not only parses the necessary structural information, but automatically includes also references to the HTML files produced by *Javasrc* and *Javadoc*. This allows SHriMP to present hyper-linked source code in the boxes of its views, along which again navigation is possible.

Simplicity: In contrast to the multi-window approach of *Rigi*, SHriMP is the code name for its own approach: *Simple Hierarchical Multi-Perspective*. Studies for evaluating these two contrasting interfaces resulted in the awareness that both of them have advantages and disadvantages. Ideal was including the ability to seamlessly switch between the two interfaces [STOR 96]. SHriMP presents a system similar to *Rigi* in nested boxes whereas diving and popping is implemented as animated zooming in always the same view pane. Once having understood the controls, it is easy to explore a system. The possibility to quickly zoom-out helps in keeping the overview, also when navigating from one node to another node, the interfaces animates this "journey" by first zooming-out, and then zooming-in again to the specific position. Operations are done with the mouse or over the menus. A *Hotbox* is provided that similar to a TV remote offers the most frequent commands in a pop-up window. The additional window *Navigation View* shows the overall picture of objects and the current position, optionally with Sugiyama or Radial layout.

Navigation between Tool States: SHriMP provides ways to navigate between tool states in form of *Back*, *Forward*, and *Home*. These steps are based on modifications in view, layout, or scale, but not on manipulations like a changed selection, a moved object, or applied filters. Double-clicking a box opens or closes its internals. Right-mouse-button-clicking is used to zoom-in, if SHIFT key is pressed, to zoom-out. The tree modes for zooming are *Zoom* which is pure step-less zooming; *Magnify* which zooms to the selected box to optimally fill the window; and *Fisheye* [FURN 86] which keeps the current point of view and only enlarges or minimizes the size of the selected box. A search utility provides the possibility to find boxes according to their name and their type. Regular expressions are accepted as search terms. Interesting views can be added to *Filmstrips*. Boxes can be copied, cut, and pasted.

Navigation in Graphs: References of boxes in the widgets of other boxes are represented as hyperlinks. These widgets are hypertext documents in the case of

the boxes are representing source code, but can be different widgets in the case of e.g., a clinical information system about Diabetes. Navigation along arcs (edges) is possible manually, by following the "arrow". Sub boxes of a box can be arranged with the help of four layout algorithms. First by a grid layout, sorting by number of children, by number of relationships; by type of node, or in alphabetical order; second, by a radial layout; third, by a parameterizable spring layout [Gius 99]; and fourth by Sugiyama trees [Sugi 81].

Navigation in Object-Oriented Models: Since the widgets in "leaf boxes" are Javadocs, as presented in Section C.2.2, the semantic navigation along references is once the same as for Javadocs. The structure of a system is modeled with the concept of nested boxes, that can be navigated as described above. Independent from the currently displayed level of detail of a box, the arcs are pointing to boxes with related information. The nodes types in a sample Java project include *Class, Interface, Global Variable*, or *Method*. The interconnecting arcs include *access, call, extends*, or *inFile*.

Efficiency: Once familiar with the controls of SHriMP, it is possible to fast browsing a system, and jumping from one view to the other. Thanks to the *Back* button, a short drift into a blind alley is usually not a big loss of time. The possibility of recording *Filmstrips* helps in relocating previous points of interest.

Feedback: Nodes can be annotated, and the attributes can be edited. The views can be interactively manipulated, boxes can be moved to the desired position.

Classification: An explicit rating is made by the user, when adding a view to the *Filmstrip*. This means that the users thinks he can use this view at a later time of moment to get some information. The spring layout considers connecting arcs to measure the proximity of nodes.

Complexity Reduction: By interactively manipulating a system, opening and closing boxes, and by magnifying and displaying them in the right format, can result in expressive views. Information of a certain kind can be hidden by filtering. The *Navigation View* presents an overview of hidden and not hidden parts of the system in different ways, as described in the paragraph about simplicity. Layouts like the spring can help to identify logical coupling in contrast to the physical coupling of source code artifacts, based on which SHriMP initially splits a system.

Memory: SHriMP keeps the complete information about a subject system in a *Project*. A project contains the RSF graph including the user's annotations, and it contains the whole "sessions" of views, which the users has created. Finally a project contains also the *Filmstrip*, a collection of views, that the users considered like a kind of bookmarks.

Storage: Single views can be saved and reloaded later. Views can be used for further computation as JPG images. *Filmstrips* can also be saved and loaded. Finally whole projects can be saved and loaded.

Extensibility: SHriMP can be extended on various levels. First, the RSF model allows defining new semantics. Second, the *Java Extractor* can be customized to generate models of different levels of detail. Third, the user interface can be extended with new widgets displayed in the leaf boxes.

C.2.5 Small Worlds

Application: Small Worlds is a commercial application for analyzing and visualizing large-scale software, developed by the Information Laboratory.

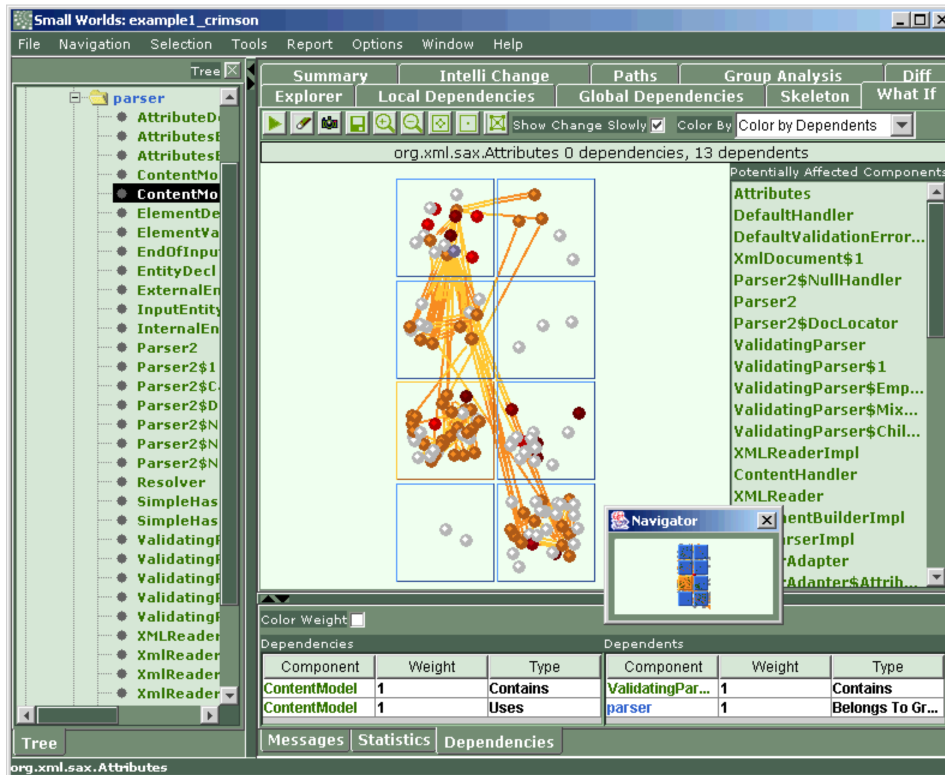


Figure C.5: Screen Capture of Small Worlds.

Description: Small Worlds is a tool for visually analyzing programs source. It provides a visual explorer and tools for checking global and local dependencies - for Java and C++. The Desktop is partitioned into *left area* and *central area*. The left area contains navigation controls, which allow users to browse through the views present in the central area. Both areas host tab-based views. The individual *tabs* are like plug-ins. Three versions of Small Worlds are created in response to their suggestions and inquiries. The *Basic Edition* includes *explorer* and *analysis* tabs, it is oriented towards individual developers working on small projects. The *Professional Edition* comes with, among other tabs, *Skeleton* and *What-If*; it is geared towards engineers and senior programmers. The *Enterprise Edition* in-

cludes tabs for managers, *Reports, Charts, Statistics, Diff, Change Planning*, et cetera. Small Worlds is available for Microsoft Windows, for Sun Solaris, and for the Linux platform.

http://www.thesmallworlds.com/

Relations: Recently a module for the integration into Sun Microsystems' Forte for Java Integrated Development Environment (IDE) has been released.

Information Laboratory announced that it will add C#, C, and Visual Basic support to its Small Worlds, and finally the application will be able to cope with relational databases, tracking and analyzing the dependencies among all database objects, tables, views, and stored procedures.

Low Entry Barriers: Small Worlds promises reduced learning curve for new team members, a simplification of the knowledge transfer process, the ability to identify potentially problematic parts of the system by showing bird-eye views of your software. We believe that this can be true. An online tutorial guides you through the main tabs of the system and shows the tool in action. After having seen this tutorial you should know enough to start the tool and run an example analysis. In the user guide in the Portable Document Format (PDF), all relevant features and concepts are explained. The help menu links to additional information about *Metrics* and *Mathematical Formulas*. There is an *Application Guide* showing the installed *tabs* and giving instructions on how to use, and there is a entry *Notations* that links to a documentation of utilized graphs, figures, and symbols.

Completeness: Small Worlds integrates a lot of tools in one application. The navigation area on the left hosts the tools *Tree, Groups History, Find, Missing, Diagrams*, and *Inheritance*. The view area in the center hosts the tools *Explorer, Local Dependencies, Global Dependencies, Skeleton, What-If, Summary, Dependencies, Statistics, Intelli-Change, Paths, Group Analysis*, and *Diff*. We discuss the functionality of the individual tools along the concerns they address.

Simplicity: At first sight the application looks complicated. However, this is because Small Worlds integrates many powerful tools into one application. Each individual tool is simple and intuitive to use. Though, time is needed for learning to understand the views with the utilized colors or metrics.

Navigation between Tool States: The possibilities of navigation between tool states differ, according to the currently used tab. In the *Explorer* you can navigate through your system. It supports going *Back* and *Forward* in the list of visited elements which is simultaneously maintained by the *History* navigation tab. *Zooming* is also possible, like in the *Skeleton* and *What-If* tabs. The skeleton tab arranges the system's components in a grid, with the most independent components on the bottom. The skeleton tab can help to gain insights into the specific stability issues. The what-if tab visually displays the impact of potential changes.

Navigation in Graphs: A variety of views provide graphical representations of groups of elements, connected by relations. In the explorer and in the what-if tab, classes are shown and connected by various kinds of dependencies.

Navigation in Object-Oriented Models: Small Worlds basically supports navigation along classes, dependent classes, interfaces, packages, groups, components and APIs. It shows the classes' method signatures, but not the source. Neither navigation was possible on method level, though the information must be there, since it is used by the tool to make the impact analysis.

Efficiency: In the detail tab presented at the bottom of the central area, detailed information about the currently selected elements can automatically be displayed. Most of the views can directly be saved as images. The predefined charts and statistics provide a step towards a structured reverse engineering process. Additional tools can help you to automate your tasks. *Diff* is used to compare old and new snapshots of your applications. *Intelli-Change* determines the best change order and parallel step changes when you need to modify components in your system in the fastest possible way. *Paths* lets you determine from two particular components whether they are connected or not.

Feedback: Small Worlds is a read only analysis instrument, without the possibility to enter any additional information.

Classification: A concept of dependency is used to determine the impact of changes which is quantified by its *weight*. The lists of dependent components can be sorted according to their *type*, name or weight. Types include *Uses, Belongs To Group*, and *Extends*, thus the traditional relations between object-oriented source code artifacts.

Complexity Reduction: In the explorer view the *Degrees of Separation* can be set. This defines the level of detail of the current consideration and the depth for searching dependencies. Different types of components and relations have different colors and shapes. Like that they are easily recognizable. Predefined charts like the *Distribution of Local Dependents* help to identify further anomalies in components.

Predefined reports lead to complexity-reduced statements like “The overall stability of the system is 90%”. We believe that such statements are meant to be considered as a relative value to be compared to other systems measured by the same tool, or the same system in another version to analyze the evolution. The menu entry *Report* provides additional short-cuts to several predefined dependency analysis reports. Finally, the *Group Analysis* can help in understanding the logical and physical organization of components.

Consistency: Before a change is made it can be simulated by the what-if tool. The results of the impact analysis is visually represented. Additionally lists of dependent components are created which can be exported in HTML format. A small *Navigator* child window is available for the explorer, skeleton, and what-if tabs. This navigator child window shows a tiny overview of the closure of the graph displayed in the center area.

Memory: A history of selected components is kept. Navigation along this list is possible. A list of recent projects is kept.

Storage: *Charts*, *Statistics*, and *Summary* are tabs that can be saved in HTML format. The statistics include numbers or percentages like *Number of Components*, or *Clustering*. The summary is a detailed report about the system and its characteristics. All the views can directly be saved as pictures. The data sets in form of table in the tabs *Global Dependencies*, and *Local Dependencies* can be saved in HTML format.

Extensibility: Controls and tabs can be hidden or shown. The subareas of the application can be masked.

C.2.6 TheBrain

Application: TheBrain is a commercial tool for managing information by visually organizing resources and relations. It is developed by The Brain Technologies.

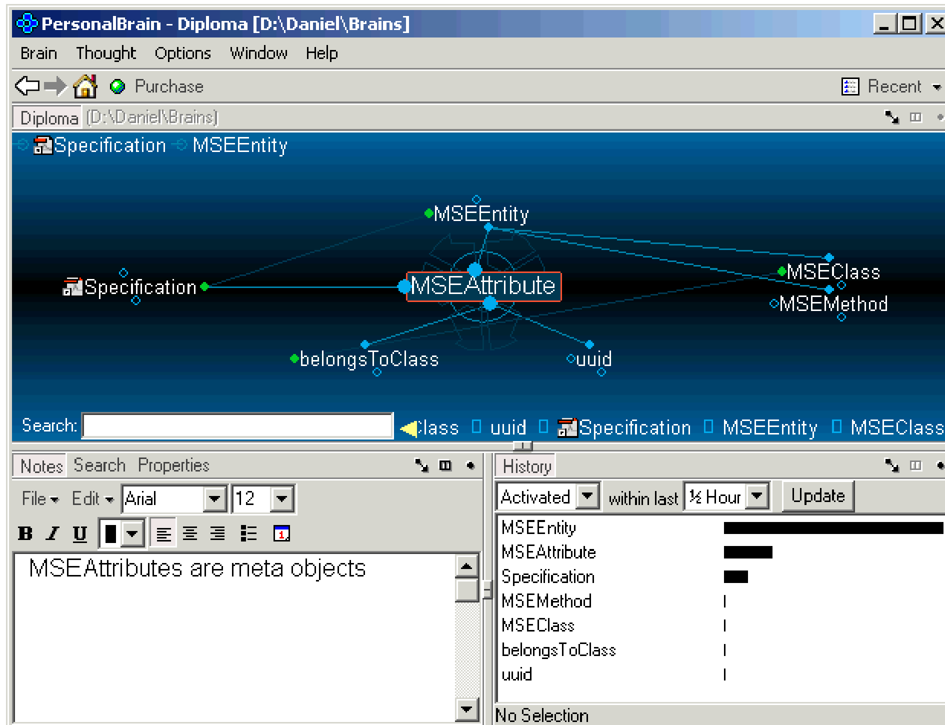


Figure C.6: Screen Capture of PersonalBrain.

Description: The visible area of a *brain* is called the *plex*, a collection of closely related *thoughts*. In the topic map terminology, a thought is a topic. The *active thought* is positioned at the center of the plex. Thoughts are connected by *relations*. Relations are equivalent to the topic map concept of associations. Related thoughts are linked visually through edges that end in one of the three circles around the active thought, called *gates*. *Zones* are areas in the plex devoted to specific relationships of the active thought. There are four zones: the *parent zone* above the active thought, the *child zone* below it, the *jump zone* to its left, and the *sibling zone* to its right. To activate a thought, a mouse click on the thought is needed. Thoughts can have occurrences in form of links to documents, to web sites, or to any other piece of resource. Such occurrences are called the *content* of a thought. The Brain

is available in different editions: *BrainEKP* is an enterprise knowledge platform for connecting and navigating multiple information systems across departments. *PersonalBrain* is the edition we tested, running on Microsoft Windows platforms. *SiteBrain* is a Java applet with the *plex* user interface paradigm of PersonalBrain for web site navigation.

http://www.thebrain.com/

Relations: *WebBrain* is a web site that lets you search the web visually. The information pool behind *WebBrain* is the Resource Description Framework (RDF) directory, maintained by the *Open Directory Project*[8].

BrainSDK is the Software Developer Kit (SDK) based on Java version 1.1 that enables partners to embed TheBrain interface and visualization capabilities in other applications. With this SDK, also bridges can be built to standard topic maps (Section 2.1) and repositories in other meta data formats.

Low Entry Barriers: TheBrain provides a simple and intuitive user interface. When visiting the product web site, you already use *SiteBrain* when navigating the site. Five online tutorials demonstrate the key features in action. A glossary is presented in case of confusion about the names and concepts. Alternatively to the *Flash* tutorials, a user guide in PDF can be downloaded. After the simple installation also a integrated help is available. *Hints* and a *Tip of the Day...* further help to learn more about the tool. A *Getting Started Wizard* guides you to define your own brain, collecting information about your family, friends, hobbies et cetera. Once you have created such a brain about a domain which you know very well, you quickly learn how to use the tool.

Completeness: TheBrain is primarily a user interface, a navigation tool for many different kinds of models. Among the models is the World Wide Web /WWW), the file structure of your hard disk, or other models of knowledge that evolve when you create your thoughts and relations. The features for navigation are concise and clear.

Simplicity: After having seen the tutorial, it is easy to work with the tool. The concepts for operating the tool are intuitive, and the representation of information is simple and consistent. The four types of relationships are easy to understand and remember. Searching thoughts in an alphabetical list or in the history list works like users are used to.

[8] http://dmoz.org/

Navigation between Tool States: A tool state in TheBrain is characterized by the currently active thought. Navigation along the history of active thoughts is possible in two ways. First, going *Home*, *Back* and *Forward* in the list. Second, by selecting a past active thought in the *History*. *Undo* and *Redo* of most actions like rename, or forget a thought is possible. Another feature of TheBrain is *Wander* which plays a filmstrip of wandering along the path of previously activated thoughts.

Navigation in Graphs: The plex of TheBrain always displays an active thought, and related thoughts connected by relations. Since the relations are grouped and arranged according to the type of relation to the active thought, navigation is primarily done by series of activating thoughts and related thoughts.

Navigation in Object-Oriented Models: We prototypically created a brain with some example thoughts representing source code artifacts. At the first sight it seemed to be possible to represent a whole reverse engineered system in TheBrain. The mapping from the object space to TheBrain's data model which is a proprietary form of topic maps (Section 2.1). However, effort would be needed to write a bridge from a specific meta model to TheBrain. As far as we could see, one thought can only have a maximum of one occurrence. This would set certain limits, especially in the context of meta meta modeling. Another form of navigation that goes beyond the scope of TheBrain is the feature *Search Web...* which takes the name of the active thought as the search term for a directory or search engine within a configurable set. Although, the results of such a search are presented in the web browser. By dragging the mouse from a thought to the web browser, a relation is created that links to the specific Uniform Resource Locator (URL).

Efficiency: TheBrain can be configured for different ways of arranging its child windows *Notes, Search, History*, and *Properties*. These windows can be docked, or floating. They can be attached to each other, or divided. Depending on your preferences you can display detailed information about the active thought in the corresponding child windows. If you find the way from one thought to another to be too long, you can get an abbreviation by creating a direct relation between two thoughts. Several utilities help for example in the case when you want to publish a local brain on the Internet. Among them are *Convert Files into Shortcuts*, *Convert Shortcuts into URLs*, *Convert URLs into Shortcuts*, and *Convert Search and Replace URLs*. Folders and files can be imported automatically in form of links to the corresponding items. A parser for *Netscape* bookmark files is also provided.

Feedback: TheBrain automatically tracks and logs the activations of thoughts. Thoughts can be annotated and renamed. By connecting thoughts to resources, their content type is made explicit. Thoughts can be classified to certain criteria, as described in the next item.

Classification: The *content type* of a thought is the type of resource of its occurrence. There are predefined types, e.g., the file types of your operating system, but also folders and URLs. Relations are classified in the four types *Parent, Child, Jump,* and *Sibling*. In the history list, thoughts are sorted according the *relative length of time they were active*. This value is illustrated by a bar chart. This *intensity* is used as a measure for sorting the thoughts according to their speculated importance. Thoughts can be kept *private* or *public*. Additionally this scope can be made variable by rules taking *before*, or *after*, and a certain *date* as arguments. Before thoughts are deleted, they can be set to *forgotten*.

Complexity Reduction: Forgotten thoughts can be hidden or shown. In the search field, auto completion helps you in pre-selecting lists of pattern matching thoughts. An advanced search tool allows searching for strings in names, keywords, notes and content. Alternatively the alphabetical list of all thoughts can be considered, or filtered by type of content and some predefined more complex types. Among the latter are *Related Thoughts*, taking the number of generations as argument; *Parentless Thoughts*, *Forgotten Thoughts*, or *Invalid Web Links*. The history can be filtered by time of last *Modification, Creation,* or *Activation*.

Consistency: The layout for thoughts is consistent according to their type to the active thought. The history remembers where you are, and lets you find back. Undo mechanisms provide a way to reject actions with unintended effects.

Memory: *Pins* are bookmarks that can be put at the upper area of the plex. The list of the most recent thoughts, called *Past Thought List*, is presented at the bottom area of the plex. Additionally you can switch the whole context by selecting an item in the list of *Recent* brains. As mentioned before, a history of actions and active thoughts is kept.

Storage: One particular brain consists of a project file plus a folder with the corresponding resources. A plex can be printed. Brains can be published to the Internet, and thus shared with other people. Conversely brains of other people can be imported to increase your knowledge pool.

Extensibility: Content types can be added and self-defined. The animation of the user interface can be customized by several parameters. The effect for double-clicks with the mouse can be changed. Additional third party directories and search engines can be accessed via the *Search Web...* feature. Advanced extensions need the SDK to be used.

C.2.7 Together

Application: Together ControlCenter is a class modeling and programming environment, keeping source and model diagrams in sync. It is a commercial product developed by TogetherSoft.

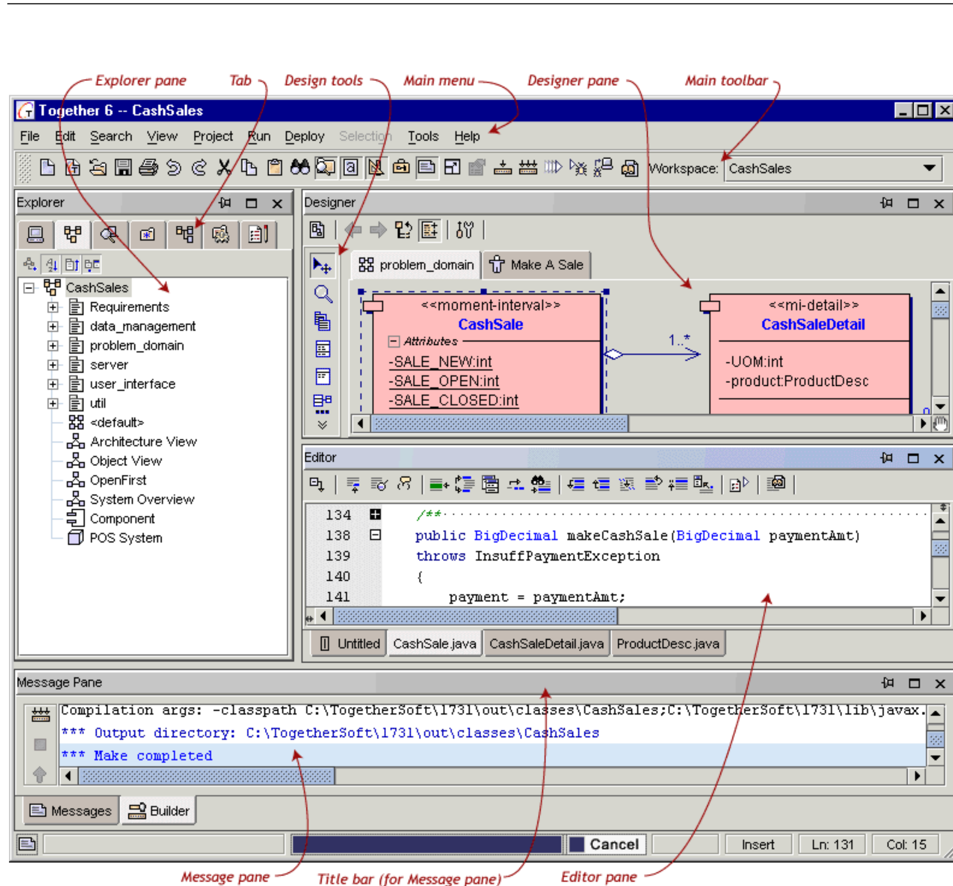


Figure C.7: Screen Capture of Together ControlCenter.

Description: Together is a communication link among analysts, designers, developers, and programmers. The design can visually be modeled in the Unified Modeling Language (UML). Together includes a complete IDE, a graphical user interface builder, a testing framework module, and a debugger. The work is organized in *projects*. Each project can be viewed by several *workspaces* which are pre-definitions for arranging the various *panes*. The main five panes are *Explorer pane, Designer pane, Inspector pane, Editor pane*, and *Message pane*. Together supports four basic *roles*: *Business Modeler* – diagram editor central; minimal

menus for simplicity's sake. *Designer* – diagram and text editor central; everything up to the point of compilation. *Developer* – diagram and text editor central; compile, debug, assemble, deploy, and run. *Programmer* – text editor central; compile, debug, assemble, deploy, and run. The supported programming languages are Java, C++, IDL, Visual Basic, and C#. Versions are available for many platforms like Microsoft Windows, Sun Solaris, Linux, Hewlett-Packard HP-UX, and Compaq Tru64 UNIX.

http://www.togethersoft.com/

Relations: The *Together Community* is a virtual gathering place for Together users to post a question, read other users' postings, search for useful information, and make your own *contribution*. The contributions are downloadable from categories like *Patterns, Modules/Plugins*, or *Configs*. Special interest forums exist, e.g., for UML, OO Design, WebServices or Java.

Together supports *CVS* and *ClearCase* as Version Control Systems (VCS). Other generic VCSs can be configured.

Low Entry Barriers: While downloading the 70 MB installation files, there is time to visit the *Practical Guide to Getting Started with Together ControlCenter* which is an online tutorial, concisely guiding through the concepts and features of the application. After the installation, the tool starts up with a *Tip of the Day* window. A wizard is provided to setup e.g., the initial role, to select an example project, or to create a new one. Additional information is accessible in the *User Guide* in PDF, or by the *Context Help* available from an entry in the applications help menu.

Completeness: Together is a tool that supports tasks from modeling to deployment. With the integrated designer you create UML diagrams. The design is implemented, test-run and debugged by the IDE which also supports twelve standard refactorings. The code is tested by the testing framework, and finally deployed - all from within one application. A *Javadoc*-like (Section C.2.2) HTML documentation can be generated in addition to the verbose design reports in PDF. There is automation support for various audits. More than fifty different metrics can be run over the project. A query-based search with regular expressions helps you to find diagrams and artifacts. The deployment of projects as Enterprise Java Beans (EJB) or as WebServices is possible. To setup and configure a database, the *DBMS Server Connection* admin tool is available.

Simplicity: Since the model and code are always in sync, no tedious and error prone tracking of changes is necessary. For the different roles of project members, different predefined workspaces are provided, hiding less important information, and showing what is important to that specific role.

Navigation between Tool States: Selecting a certain workspace is the shortest way to a configuration for arranging panes on a desktop. The predefined workspaces are *CodingWorkspace, DesignWorkspace*, and *DebugWorkspace*. In the designer pane zooming is possible. A history is kept for being able to jump back and forward in the list of previously selected artifacts. Bookmarks can be set in the source code. *Undo* and *Redo* of changes is supported.

Navigation in Graphs: In the designer pane, navigation along arrows of UML diagrams is possible, which stand for *Generalizations, Implementations*, general *Associations*, state *Transitions* et cetera ([FOWL 97]). Selecting an element in the explorer pane shows its diagram and source in the designer and editor panes. Conversely, when you create a new file in your project (source code or diagram), it shows up in the explorer pane. Additionally navigation from a diagram to its parent diagram is possible.

Navigation in Object-Oriented Models: Besides the graphical navigation in the diagrams and the ability of selecting artifacts from within the tree view in the explorer pane, the feature *Browse Symbol* searches for all occurrences of a given symbol in the system. With the current focus on a method, navigation to the super method is possible. Often used blocks of source code can be declared as *snippets*. There are several default snippets for common constructs in each supported language: *if, for, while*, et cetera. Jumping from one snippet to the next is possible. The menu entry *Override/Implement Method* opens a wizard for creating a new method in another class. *Show Ancestors* and *Show Descendants*, performed on a class displays the hierarchy of the current class with its superclasses, or subclasses respectively. the same menu entries, performed on a method show overridden methods, or extenders. Additionally, the *Implementing Classes* can be found for a certain method. The explorer pane shows the project and its elements. The provided tabs are *Directory, Model, Servers, Favorites, Diagram, Components, Modules, Xplorer, UIBuilder,* and *Testing*. By selecting elements in the navigation pane, the diagram pane shows the corresponding diagrams, and vice versa. In the editor pane, code artifacts can be selected. The *Select in Diagram* command searches for an appropriate UML diagram, where it automatically selects the given artifact. Conversely by selecting methods or classes in a UML diagram shows the corresponding source in the editor pane. *Search for Usages* provides another mechanism for navigating along related artifacts. In hyper linking to internal or external artifacts and information, linking any diagram or element to any external document, Uniform Resource Locator (URL), or any other element in the project

is possible. Hyperlinks are automatically generated from diagrams or elements to the HTML documentation.

Efficiency: While modeling a system in UML, the structural code for the corresponding classes, methods, and attributes is automatically generated. Bookmarks, auto completion, syntax highlighting, tool tips for method signatures and parameter types, as well as automatic source code formatting further increase the productivity in writing programs. *Surround With... [a certain snippet])* saves time of writing and minimizes misspelled words and other syntax errors in the source code. *Comment and Uncomment* of passages in the source code is possible.

Feedback: The continuous feedback of changes in the source to update diagrams and vice versa is performed in the background. While changing the code, the system automatically parses the new source and displays warnings or errors. Bookmarks can be defined, and managed globally.

Classification: A first classification of what the user sees and what is hidden is to specify the role of the current user. Additionally the various workspaces are an instrument to switch between configurations for arranging the panes according to a specific task. In the designer pane, types (class, method) and kinds (public, private) of artifacts can be shown or hidden. In the editor, a number of lines, a method's implementation, or comments can be collapsed or expanded.

Complexity Reduction: Depending on the current task information can be hidden or emphasized. Switching between diagram and source mode can help to have the currently appropriate style of displaying information in a way that it is easier to understand.

Consistency: The model and source code are always in sync. After changing the focus from one class to another, the source is compiled and saved. Soon, the use of a VCS can be an advantage, to be able to undo unwanted changes, since there is no extra action necessary to save a new version of a Java class. The *Diagram Overview* shows a small window with all the current diagrams in the designer and displays a black shadow for the visible area.

Memory: Workspaces save the current positions of panes on your desktop. On closing Together the current workspace is memorized and opened again on the next start up. The system keeps lists of *Recent Files*, and *Recent Projects*. A history remembers the visited artifacts in a system.

Storage: A project can be exported to XML Metadata Interchange (XMI) files. The whole design can be printed as a design report including the *Package Requirements, Activity Diagrams, Use Case Diagrams* with notes and hyperlinks, or *State*

Diagrams. The source can be documented in HTML. Single diagrams can be saved as images, or printed. All reports can alternatively be printed to a file in PDF.

Extensibility: *Import Model from XMI File, Import J2EE Archive*, and *Import Database Schema* are predefined automation features for importing external data. Besides of customizing the reports, many feature of the panes and tools can be customized. For further extensions and adaptations to third party tools the Application Programming Interface (API) of Together has to be used. It is fully described in a HTML documentation.

C.2.8 Additional Features

Since we could not discuss all the interesting tools we know or had a look at, we present here some additional outstanding features separately.

- **3D Views.** For creating multi-purpose 3D views, we suggest using **Visualize it!**. It produces the nicest and most colorful visualizations.
- **3D Animation.** The only tool we have seen that generates animated 3D views is **Imagix**.
- **Refactoring.** In the category of IDEs and round trip engineering tools the number of supported refactorings increases steadily. An outstanding big number of refactorings is provided by **IntelliJ IDEA**.

C.3 Summary

The simplest navigation features are *Back, Forward*, and a concept for hyperlinks. These features are available in every Internet browsers and finally begin to get also a matter of course in source code and graph browsing tools. *Undo* and *Redo* of actions is important for the consistency in a tool, however it is first often complex to reason about what exactly the original action was, and second it is hard to find the reversal of many actions. All of the tools help in reducing the complexity. Still, navigation along semantically related artifacts is often complicated.

In the category of IDEs we observe a trend towards broad refactoring support. A prerequisite to refactorings is to gather the structural information, and building up a complete parse tree. This has first be done in the VisualWorks Refactoring Browser developed by Brant from the University of Illinois at Urbana-Champaign, USA. The features are now adapted by major IDE manufacturers, fortunately they are becoming a matter of course. The implication of refactoring support is, that all the necessary structural information is available also for navigation along.

UML modelers and round trip engineering tools provide ways of generating source code from graphical representations and vice versa. With the latter they also provide some sort of reverse engineering features. We conclude that more and more reverse engineering features are integrated into forward engineering tools. The categories slowly melt. Good development environments of the future support the complete software lifecycle of reengineering / maintenance - forward and reverse engineering.

We present an overview of the discussed tools and their features in Table C.7.

Legend for Table C.7

$\ominus$	Not supported / missing - Insufficient
$\sim$	Supported with reservations - Sufficient
$\surd$	Satisfactorily supported, implemented, available - Good
N/A	Not Applicable or Unknown

Concern	*Paradigm*	Eclipse	Javadoc	Rigi	SHriMP	Small Worlds	TheBrain	Together
1.	**Clear Benefit**	√	√	√	√	√	√	√
	Installation Support	√	√	√	√	√	√	√
	Habitual Look & Feel	√	√	∼	∼	√	√	√
	Demo	⊖	⊖	√	√	√	√	⊖
	Assistants & Wizards	√	N/A	N/A	N/A	⊖	√	√
	Help	√	√	√	√	√	√	√
	Safety	√	√	√	√	√	√	∼
2.	**Complete Model**	√	∼	∼	∼	∼	∼	√
	Appropriate Features	∼	∼	∼	∼	∼	∼	∼
3.	**Simple Operation**	⊖	√	∼	∼	⊖	√	⊖
	Simple Views	⊖	⊖	√	√	∼	√	⊖
4.	**Tool State History**	√	∼	√	√	√	√	∼
	Manipulate Tool States	√	√	√	√	√	√	√
	Actions History	√	⊖	⊖	⊖	⊖	√	√
5.	**Neighborhood**	⊖	⊖	√	√	√	√	√
	Hyperlinks	∼	√	√	√	∼	√	√
	Graph Layouts	⊖	⊖	√	√	∼	√	∼
6.	**Semantic Navigation**	√	√	√	√	√	√	√
	Affinity	⊖	∼	√	√	√	√	∼
7.	**Pushing Information**	√	√	⊖	⊖	√	√	√
	Pulling Information	√	⊖	√	√	√	∼	√
	Near Features	∼	∼	∼	∼	∼	∼	∼
	Automation	√	⊖	√	∼	⊖	⊖	√
8.	**Manual Feedback**	√	⊖	∼	∼	∼	∼	√
	Automatic Feedback	√	⊖	⊖	⊖	⊖	⊖	√
	Round Trip Engineering	⊖	⊖	⊖	⊖	⊖	⊖	√
9.	**Explicit Classification**	√	⊖	√	√	√	√	√
	Implicit Classification	⊖	⊖	∼	∼	∼	⊖	⊖
10.	**Meaningful Representations**	∼	∼	√	√	√	√	√
	Reducing Volume	∼	⊖	√	√	√	√	√
11.	**Geographical Consistency**	N/A	N/A	√	√	√	√	√
	Overview	N/A	N/A	√	√	√	⊖	√
	Consistent User Interface	√	√	∼	∼	∼	√	√
	Consistent Content and State	√	√	√	√	√	√	∼
12.	**Recovery**	√	√	√	√	√	√	√
	Model Enrichment	√	⊖	√	√	⊖	√	√
	Team Support	√	√	N/A	N/A	N/A	√	√
13.	**Safety**	√	N/A	√	√	⊖	√	√
	Exchangeability	√	N/A	√	√	⊖	∼	√
	Reports	√	N/A	⊖	∼	√	⊖	√
14.	**Extend Model**	√	⊖	√	√	⊖	⊖	⊖
	Extend Tool	√	⊖	∼	∼	⊖	⊖	⊖
	New Concerns	√	⊖	∼	∼	⊖	⊖	⊖
	Customizable Export	√	⊖	⊖	⊖	⊖	⊖	∼

Table C.7: State-of-the-Art Features Overview.

Appendix D

MooseNavigator Implementation

We introduced the features of *MooseNavigator* in Section 5.2. This appendix is about the implementation details.

MooseNavigator is written in Smalltalk using the VisualWorks 3.0 / ENVY 4.0. Subclassing, extending and overriding the base classes of *CodeCrawler* is a flexible customizing procedure for creating prototypes, without embellishing the core application or endanger the core to harm in any form. This is what we did in the development of *MooseNavigator*. See Appendix A for more details about *Moose*, and *CodeCrawler*.

The multiple windows interface follows the Model-View-Controller (MVC) paradigm, where the sub windows register themselves in the dependents list of the parent application. Doing this they automatically become receiver of update notifications.

A description of some important classes of the application follows:

```
CodeCrawler subclass: #MooseNavigator
   instanceVariableNames: 'toolBar systemOverviewer
descriptionViewer previouslyVisitedNode previouslyVisitedEdge
sessionViewer logProcess'
```

MooseNavigator is the main window and central class of the system. Technically it is a subclass of *CodeCrawler*. *toolBar* holds the button pane, where actions to the session are passed down to the model, and filtering and zooming actions are directly performed on the current view. Search is in fact diving to the model subset which conforms to a given block, printing creates a postscript file of the current view. *systemOverviewer, descriptionViewer, and sessionViewer* hold the sub windows. *previouslyVisitedNode and previouslyVisitedEdge* help for remembering the focus and when it changes. In *logProcess* we keep a separate process that

writes navigation logs to the hard disk after a certain amount of time. One of the most important methods of *MooseNavigator* is *#nextPutViewWithConfiguration: aConfiguration onModelSubset: aModelSubset selectEntities: aCollectionOfEntities* which creates a new view, puts it in a new session state, adds this one to the current session, and finally displays the new view. The application is started by sending *#open* to the MooseNavigator class.

```
Object subclass: #MNUtility
   classInstanceVariableNames: 'standardFilterLibrary'
```

MNUtility is the container for constants and functions that must be globally available. It is not instantiated. We use the meta class only. It holds the central *standardFilterLibrary*. It is further used to compose description texts following a certain policy, which are then displayed in the description viewer and other places.

```
CCDrawing subclass: MNDrawing
```

MNDrawing is kept in the main application as value of the attribute *drawing*, the model of *drawing* in turn is the main application itself. MNDrawing fetches mouse clicks on figures and on the drawing - to track navigation, and pops up a dynamic context menu when the users performs a right mouse click on a figure.

```
ApplicationModel subclass: MNSessionViewer
   instanceVariableNames: 'parentApplication session
sessionTreeList selectionInSessionTreeList'
```

MNSessionViewer adds itself to the dependents list of its *parentApplication* as well as on that from its model, the *session*. Its *sessionTreeList* and *selectionInSessionTreeList* handle the tree widget and the current selection.

```
CCAbstractSubcanvas subclass: #MNDescriptionViewer
   instanceVariableNames: 'text'
```

MNDescriptionViewer adds itself to the dependents list of its *parentApplication* which it inherits as a subclass of CCAbstractSubcanvas. It holds only one extra attribute *text*, which keeps the description to show.

```
DrawingEditor subclass: #MNSystemOverviewer
   instanceVariableNames: 'parentApplication'
```

MNSystemOverviewer is a subclass of DrawingEditor - a class which belongs to the graphical framework HotDraw [BRAN 95]. Its only instance variable is *parentApplication* which stores the information to which main window the tool belongs.

```
ReadWriteStream subclass: #MNSession
   instanceVariableNames: 'name timestamp actions wrappers'
```

MNSession is as a subclass of ReadWriteStream and stores the behavior of a user. In its *collection* (the attribute is inherited from PositionableStream) it puts all the states of the tool (MNSessionState). MNSession stores also the *actions* which lead to specific states. In *wrappers* it holds the current set of MethodWrappers that is used to track these actions in detail. A MNSession further has a *name* and a *timestamp*. MNSession is also responsible for reporting - it can for example be asked for *#streamWithActionsAndStates*, which generates a report as seen some sections above. Sessions are kept in the model.

```
Model subclass: #MNAction
   instanceVariableNames: 'name session receiver message timestamp'
```

MNAction is a subclass of Model. The instance variables are *name*, *timestamp*, *session*, *receiver* and *message*. The *session* object stores the information to what session the action belongs. The *message* is a ByteSymbol with the name of the message that was sent for this action (e.g. *#crawlSystemComplexity*). The *receiver* is the object that received this message (e.g. *MooseNavigator*).

```
MethodWrapper variableSubclass: #MNMethodWrapper
   instanceVariableNames: 'session'
```

MNMethodWrapper is a subclass of MethodWrapper [BRAN 98]. Aside from the introduced instance variable *session* the only implemented method is *#afterMethod*. The MethodWrapper concept is the following: MethodWrappers are installed with: *#on: selector inClass: class*, which puts the wrapper at the position of the selector in the system's method dictionary. The wrapper provides two abstract methods: *#beforeMethod* and an *#afterMethod*. In our case, we cause the wrapper, after invoking the method itself, to add a MNAction to the wrappers *session*, containing the specification of the called method. All the wrappers in the system are removed on closing *MooseNavigator* and again installed on launching. Editing the source of wrapped methods may cause problems with consistency.

```
Model subclass: #MNSessionState
   instanceVariableNames: 'name session timestamp modelSubset view
filters selections annotation'
```

MNSessionState is a subclass of Model. Apart from the instance variables *name*, *timestamp* and *session*, MNSessionState covers the relevant information for reproducing a certain state of the tool. In the *modelSubset* the current subset of the model of entities is stored. The variable *view* keeps the complete view in memory, for performance reasons. When closing the *MooseNavigator* application, the views must be de-coupled, otherwise the garbage collector can not work properly. MNSessionState also stores the set of currently applied filters (*filters*) and the set of currently selected elements on the drawing (*selections*). Finally a MNSessionState can be renamed, annotated, or deleted. MNSessionStates are created by the constructor *#withModelSubset: aModelSubset andView: aView andSelections: aCollectionOrNil*.

```
Object subclass: #MSEFilter
   instanceVariableNames: 'name block body scope enabled'
```

MSEFilter is created with a *name*, a *body*, and a *scope*. The *body* is the instruction to build a Smalltalk block, where *each* is playing the role of the entity to which a filter is applied. A filter can be active or not - the switch for that is the boolean *enabled*. Applied filters are stored in the corresponding session state. If a block throws a *MessageNotUnderstood*-exception the execution proceeds - the filter evaluates to *false*. With the example body '*each isPrivate*' a Smalltalk block *[:each | each isPrivate]* is created. Also nested bodies like *(each isClass) and: [each isAbstract]* can be processed. To evaluate a filter call *#isFalseFor: anEntity*, or *#isTrueFor: anEntity*

```
Object subclass: #MSEFilterLibrary
   instanceVariableNames: 'filters'
```

MSEFilterLibrary is the central repository for filters. The library is kept in MNUtility_class, where all objects can access it. The collection of *filters* can be read from and wrote to streams, and like that also from an to the hard disk.

```
CCModelSubset subclass: #MNModelSubset
```

MNModelSubset contains the current set of entities, like the CCModelSubset. Additionally it allows the user to enumerate or count the entities of certain types. A filter can be applied *#applyFilter: aMSEFilter* which removes all entities for which the filter evaluates to *true*. MNModelSubset provides also a constructor for directly

hiding certain aspects: *#newWithFilter: aMSEFilter*.

As a last category of classes we finally describe the implementation of the layouts. Here is the overview of the class hierarchy, the specific descriptions follow:

```
CCCircleLayout
  MNCircleLayout
    MNFixedSizeCircleLayout
    MNDoubleCircleLayout
```

MNCircleLayout is the base circle layout. The most generic routine to arrange nodes around a center is *#layoutNodes: aCollectionOfNodes at: circleCenter withRadius: radius*. The default radius is computed as the maximal drawing pane dimension divided by 2.5

MNDoubleCircleLayout performs above methods twice, using a small radius for the attributes, and a bigger radius for the methods. The bigger radius has the default size, the smaller one is the default radius divided by two.

MNFixedSizeCircleLayout does the same as above, with the difference that it uses a radius not based on the current window size, but based on the number of entities to display. This is to avoid overlapping nodes. The formula is based on heuristics.

Bibliography

[Dem 02] S. Demeyer, S. Ducasse, and O. Nierstrasz, editors. Object-Oriented Reengineering Patterns. Morgan Kaufmann, 2002. (pp 1, 4)

[Duc 99] S. Ducasse and S. Demeyer, editors. The FAMOOS Object-Oriented Reengineering Handbook. University of Berne, October 1999. (pp 1, 75)

[ISO 99] *ISO/IEC 13250 Topic Maps*. Industry standard, International Organization for Standardization (ISO), Geneva, December 1999. (p 14)

[OTI 01] *Eclipse Platform Technical Overview*. Research report, Object Technology International, Inc., July 2001. (p 100)

[ALPE 98] S. R. Alpert, K. Brown, and B. Woolf. The Design Patterns Smalltalk Companion. Addison Wesley, 1998. (p 4)

[ANTO 99] C. H. Antoni and T. Sommerlatte. Report Wissensmanagement. Symposion Publishing, 1999. (pp 10, 11)

[BECK 97] K. Beck. Smalltalk Best Practice Patterns. Prentice-Hall, 1997. (p 4)

[BECK 00] K. Beck. Extreme Programming Explained: Embrace Change. Addison Wesley, 2000. (p 1)

[BECK 01] K. Beck and M. Fowler. Planning Extreme Programming. Addison Wesley, 2001. (p 33)

[BEDE 00] B. Bederson, J. Meyer, and L. Good. *Jazz: An Extensible Zoomable User Interface Graphics ToolKit in Java*. In UIST 2000, New York, Mai 2000. ACM. (p 114)

[BERN 99] T. Berners-Lee and M. Fischetti. Weaving the Web : The Original Design and Ultimate Destiny of the World Wide Web by its Inventor. Harper, San Francisco, 1999. (p 13)

[BRAN 95] J. Brant. HotDraw. Master's thesis, University of Illinois at Urbana-Chanpaign, 1995. (p 136)

[BRAN 98] J. Brant, B. Foote, R. Johnson, and D. Roberts. *Wrappers to the Rescue*. In Proceedings ECOOP'98, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998. (p 136)

[BROW 98] W. J. Brown, R. C. Malveau, H. W. McCormick, III, and T. J. Mowbray. *AntiPatterns*, 1998. (pp 29, 71)

[CHIK 90] E. J. Chikofsky and J. H. Cross, II. *Reverse Engineering and Design Recovery: A Taxonomy*. IEEE Software, pages 13–17, January 1990. (p 2)

[CINC 02] Cincom. VisualWorks - Application Developer's Guide, http://www.cincom.com/. 1993-2002. (p 69)

[CONK 87] J. Conklin. *Hypertext: An introduction and survey*. IEEE Computer, vol. 20, no. 9, pages 17–41, 1987. (p 4)

[COOP 95] A. Cooper. About Face - The Essentials of User Interface Design. Hungry Minds, 1995. (p 4)

[DAVE 98] T. H. Davenport and L. Prusak. Working Knowledge - How Organisations manage what they know. Harvard Business School Press, 1998. (pp 6, 8, 10)

[DEME 99] S. Demeyer, S. Ducasse, and M. Lanza. *A Hybrid Reverse Engineering Platform Combining Metrics and Program Visualization*. In F. Balmas, M. Blaha, and S. Rugaber, editors, Proceedings WCRE'99 (6th Working Conference on Reverse Engineering). IEEE, October 1999. (pp 28, 77)

[DEME 01] S. Demeyer, S. Tichelaar, and S. Ducasse. *FAMIX 2.1 - The FAMOOS Information Exchange Model*. Research report, University of Berne, 2001. (pp 19, 76)

[DUCA 00a] S. Ducasse, M. Lanza, and S. Tichelaar. *Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems*. In Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000), June 2000. (pp 50, 76, 77)

[DUCA 00b] S. Ducasse, M. Lanza, and L. Steiger. *A Query-Based Approach to Support Software Evolution*. In ECOOP'2000 International Workshop of Architecture Evolution, 2000. (p 78)

[DUCA 01a] S. Ducasse and M. Lanza. *Towards a Methodology for the Understanding of Object-Oriented Systems*. Technique et science informatiques, vol. 20, no. 4, pages 539–566, 2001. (pp 4, 33)

[DUCA 01b] S. Ducasse, M. Lanza, and S. Tichelaar. *The Moose Reengineering Environment*. Smalltalk Chronicles, August 2001. (pp 50, 76)

[FAVR 01] J.-M. Favre. *GSEE: a Generic Software Exploration Environment*. In Proceedings of the 9th International Workshop on Program Comprehension, pages 233–244. IEEE, Mai 2001. (pp 50, 59)

[FIND 79] N. Findler. *Associative Networks: Representation and Use of Knowledge by Computer*, 1979. (p 13)

[FOWL 97] M. Fowler. UML Distilled. Addison Wesley, 1997. (p 128)

[FOWL 99] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. Refactoring: Improving the Design of Existing Code. Addison Wesley, 1999. (p 3)

[FURN 86] G. Furnas. *Generalized fisheye views*. In In Proceedings of ACM CHI'86, pages 16–23, Boston, MA, April 1986. (p 115)

[GAMM 95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns. Addison Wesley, Reading, Mass., 1995. (p 4)

[GIUS 99] R. T. Giuseppe Di Battista, Peter Eades and I. G. Tolls. Graph Drawing - Algorithms for the visualization of graphs. Prentice-Hall, 1999. (pp 31, 111, 116)

[GRIF 82] R. L. Griffith. *Three Principles of Representation for Semantic Networks*. In ACM Transactions on Database Systems, 1982. (p 13)

[GRIS 92] R. Grishman, C. Macleod, and J. Sterling. *New York University: Description of the proteus system as used for muc-4*. In Proceedings of the Fourth Message Understanding Conference (MUC-4), pages 233–241, June 1992. (p 5)

[JEFF 01] R. Jeffries, A. Anderson, and C. Hendrickson. Extreme Programming Installed. Addison Wesley, 2001. (p 1)

[KICZ 96] G. Kiczales. *Beyond the Black Box: Open Implementation*. IEEE Software, January 1996. (p 5)

[KITC 97] R. Kitchin, M. Blades, and R. Golledge. *Relations between psychology and geography*. Environment and Behavior, vol. 29, no. 4, pages 554–573, 1997. (pp 23, 45)

[KSIE 00] R. Ksiezyk. *Answer is just a question [of matching Topic Maps]*. In XML Europe 2000, 2000. (p 16)

[LANZ 99] M. Lanza. Combining Metrics and Graphs for Object Oriented Reverse Engineering. Diploma thesis, University of Bern, October 1999. (pp 20, 28, 29, 49, 50, 67, 77)

[LANZ 01a] M. Lanza and S. Ducasse. *A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint*. In Proceedings of OOPSLA 2001, pages 300–311, 2001. (p 49)

[LANZ 01b] M. Lanza, S. Ducasse, and L. Steiger. *Understanding Software Evolution using a Flexible Query Engine*. In Proceedings of the Workshop on Formal Foundations of Software Evolution, 2001. (p 78)

[LEHM 92] F. Lehmann and E. Y. Rodin. Semantic Networks in Artificial Intelligence. Pergamon Press, 1992. (p 13)

[LEHM 96] M. M. Lehman. *Laws of Software Evolution Revisited*. In European Workshop on Software Process Technology, pages 108–124, 1996. (p 1)

[MCKN 91] C. McKnight, A. Dillon, and J. Richardson. Hypertext in Context. Cambridge University Press, 1991. (p 4)

[MÜLL 86] H. Müller. *Rigi - A Model for Software System Construction, Integration, and Evaluation based on Module Interface Specifications*. PhD thesis, Rice University, 1986. (p 109)

[NGUY 00] S.-T. Nguyen. *Environnements d Exploration de Grands Logiciels*. Research report, University Joseph Fourier, Grenoble, 2000. (p 98)

[NIEL 90] J. Nielsen. Hypertext and hypermedia. Academic Press, 1990. (p 4)

[NOY 00] N. F. Noy, R. W. Fergerson, and M. A. Musen. *The knowledge model of Protégé-2000: Combining interoperability and flexibility*. In 2nd International Conference on Knowledge Engineering and Knowledge Management (EKAW'2000), Juan-les-Pins, France, Berlin, 2000. Springer-Verlag. (p 114)

[NOY 01] N. F. Noy, M. Sintek, S. Decker, M. Crubézy, R. W. Fergerson, and M. A. Musen. *Creating Semantic Web Contents with Protege-2000*. IEEE Intelligent Systems, vol. 16, no. 2, pages 60–71, 2001. (p 114)

[PINT 95] X. Pintado. *The Affinity Browser*. In O. Nierstrasz and D. Tsichritzis, editors, Object-Oriented Software Composition, pages 245–272. Prentice-Hall, 1995. (p 18)

[PROB 99] G. Probst. Wissen Managen: Wie Unternehmen ihre wertvollse Ressource optimal nutzen. Frankfurter Allgemeine Zeitung; Gabler, 1999. (pp 8, 10, 12, 72)

[RASK 00] J. Raskin. The Humane Interface. Addison Wesley, 2000. (pp 4, 18, 21, 22, 23, 42, 45, 48)

[RATH 99] H. H. Rath and S. Pepper. *Topic Maps: Introduction and Allegro*. In XML Europe '99, 1999. (p 14)

[RATH 00] H. H. Rath. *Making topic maps more colourful*. In XML Europe 2000, 2000. (p 16)

[RAYS 01] D. Rayside, M. Litiou, M.-A. D. Storey, and C. Best. *Integrating SHriMP with the IBM WebSphere Studio Workbench*. In CASCON'2001, Toronto, Canada, 2001. (p 114)

[RHEI 85] H. Rheingold. Tools for Thought. The MIT Press, 1985. (p 9)

[RIEL 96] A. J. Riel. Object-Oriented Design Heuristics. Addison Wesley, 1996. (p 3)

[STEI 01] L. Steiger. Recovering the Evolution of Object Oriented Software Systems Using a Flexible Query Engine. Diploma thesis, University of Bern, June 2001. (p 78)

[STOR 95a] M.-A. D. Storey and H. A. Müller. *Graph layout adjustment strategies*. In Proceedings of Graph Drawing 1995, Passau, Germany, pages 487–499. Springer Verlag, September 1995. (p 111)

[STOR 95b] M.-A. D. Storey and H. A. Müller. *Manipulating and documenting software structures using SHriMP views*. In Proceedings of the 1995 International Conference on Software Maintenance, 1995. (p 113)

[STOR 96] M.-A. D. Storey, K. Wong, D. Hooper, K. Hopkins, and H. Müller. *On Designing an Experiment to Evaluate a Reverse Engineering Tool*. In Proceedings of the WCRE'96, Monterey, California, USA, 1996. (pp 110, 115)

[STOR 97] M.-A. D. Storey, K. Wong, and H. A. Müller. *How Do Program Understanding Tools Affect How Programmers Understand Programs?* In I. Baxter, A. Quilici, and C. Verhoef, editors, Proceedings Fourth Working Conference on Reverse Engineering, pages 12–21. IEEE Computer Society, 1997. (p 4)

[SUGI 81] K. Sugiyama, S. Tagawa, and M. Toda. *Methods for Visual Understanding of Hierarchical System Structures*. IEEE Transactions on Systems, Man and Cybernetics, vol. SMC-11, no. 2, February 1981. (pp 111, 116)

[SVEI 97] K.-E. Sveiby. The New Organizational Wealth: Managing and Measuring Knowledge-Based Assets. Berret-Koehler Publishers, San Francisco, 1997. (p 10)

[TICH 01] S. Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Bern, 2001. (pp 19, 50, 76)

[WARE 00] C. Ware. Information Visualization. Morgan Kaufmann, 2000. (pp 4, 21)

[WINT 01] A. Winter. *Exchanging Graphs with GXL.* In P. Mutzel, editor, Graph Drawing - 9th Interational Symposium, GD 2001, Vienna. Springer Verlag, September 2001. (pp 19, 82, 110)

[WONG 98] K. Wong. *Rigi Users's Manual*. Research report, University of Victoria, 1998. (pp 19, 109)

[YANG 93] G. Yang and J. Oh. *Knowledge Acquisition and Retrieval Based on Conceptual Graphs*. In Symposium on Applied Computing. ACM, 1993. (p 13)