# 3D-UMLVis – Visualizing Design Pattern in 3D Space

By

**Vu-Loc Nguyen**

A Thesis

Presented to the Faculty of Computer Science and Software Engineering of

**Concordia University**

in partial fulfillment of the Requirements for the Degree of

**Master of Computer Science**

Montreal, Quebec, Canada

February 2006

*Your file Votre référence*
*ISBN: 0-494-14332-0*
*Our file Notre référence*
*ISBN: 0-494-14332-0*

# Abstract

**3D-UMLVis – Visualizing Design Pattern in 3D Space**
**Vu-Loc Nguyen**

With the ever increasing complexity of software systems programmers face new challenges in comprehending the structures of these programs, their artefacts, and the behavioural relationships among these artefacts. While modeling languages and notations that were introduced to support the forward engineering process have been quite effective in abstracting the underlying information, these techniques have failed when applied in a reverse engineering context. One reason for the failure is the information overload due to the level of detail available at the source code and a lack of appropriate filtering and analysis techniques. Another limitation of the current approaches is the conceptual gap that exists between the models created during the forward and reverse engineering process. This occurs because the reverse engineered model cannot convey the information and justification behind the architecture chosen. This gap is particularly noticeable when one considers the differences in layout, grouping and organization between the original and reverse engineered models. This thesis attempts to addresses these issues, by introducing a 3D extension of UML (Unified Modelling Language) diagrams to support the visualization of recovered design patterns from source code. The goal is to provide additional guidance during program comprehension. In addition, new viewing techniques are proposed to facilitate the navigation and filtering of the information in the three dimensional world.

## TABLE OF CONTENTS

# Table of Figures

# Chapter 1 Introduction

Source code comprehension plays a predominant role in facilitating software maintenance and evolution. The essence of comprehension is identifying program artefacts and understanding their relationships. Software engineers typically start comprehending a program by identifying artefacts at a certain abstraction level and then aggregate this knowledge to form a more abstract understanding [STOR97]. Comprehension of software systems can be improved by providing higher level of abstraction to visualize the data to be observed and inspected. However, regardless of the visualization technique the sheer volume of information presented to the developers becomes daunting, as programs grow more complex and larger in size [MAYR98, STOR99].

Design patterns have become increasingly popular among software developers since the mid 1990s [GAMM95]. They help to communicate intent and scope of designs, help new developers to avoid problems that have been experienced in the past and to allow reuse of successful designs. Design patterns are typically modeled with Unified Modelling Language (UML) [UML] [GAMM95] at the risk of losing their purpose and role within the design since UML does not provide a direct modeling support for design patterns. The benefits of design patterns are thus significantly compromised since the domain knowledge and design decisions associated with the original patterns are no longer available. Furthermore, in the context of reverse engineering, these reverse engineering tools (covered in section 2.2.4) do not support the recovery of design patterns or their visualization at all. Reverse engineering tools focus on the recovery of UML

class diagrams (without visualizing design patterns) as their major metaphor for designs from the source code.

This research proposes to combine two techniques, design pattern recovery and 3D software visualization to enhance the expressiveness of UML with respect to visualized reverse engineered software designs from source code. The presented design pattern visualization approach in 3D provides maintainers and users with additional insights into the complex relationships among data without having to analyze the underlying source code in detail. Most importantly, it enables maintainers to better understand software design decisions, which are usually not easy or explicitly found in reverse engineered UML class model diagrams. As a part of this thesis, a prototype tool was developed for the 3D visualization. Entitled **3D-UMLViz**, the prototype [RILL05] displays design patterns with respect to the UML based layout proposed by the "Gang of Four" (GoF). **3D-UMLViz** can support cases where a class participates in more than one design pattern, commonly known as pattern cross-cutting. Since 3D interaction poses its own challenges, this thesis also experiments with new ideas to navigate in 3D space using traditional input devices like mouse and keyboard. In fact, **3D_UMLViz** adopts techniques that have been shown to be useful in the computer gaming industry. **3D_UMLViz** is part of the CONCEPT project [RILL02b] that was originally introduced to explore new program comprehension techniques and approaches. Its main goal was to assist programmers in creating mental models and comprehending software systems.

The thesis is organized as follows. Chapter 2 introduces background relevant to this research followed by the research hypotheses and research contributions of this thesis (Chapter 3). The implementation of **3D-UMLViz** and its integration to the CONCEPT

research framework is described in Chapter 4. Chapter 5 provides the conclusion and summarizes the contribution of this thesis along with a brief discussion on future work.

# Chapter 2 Background

Due to the rapid development of the software technology during the last decades and the increased demand for software products, a large number of software systems have been developed. The resulting legacy code is passed on from generations of programmers as part of a company's "know-how" and assets. On the other hand, the continuously changing needs of the business environment require those systems to be always up to date with the latest technologies and to evolve during their life cycle. Evolving often means refactoring, adding new functionalities, modifying existing ones or migrating to a new environment. However, to succeed in such a step, a full understanding of the software system is required, leading to the need to recover its architecture and design decisions and to pass on to the maintenance team. In this chapter background literature relevant to program comprehension, software visualization and 2D versus 3D software visualization is reviewed.

## 2.1 Program Comprehension

The long life cycle of software systems makes it in most cases impossible for a maintenance team to have the same members as the development team. A large body of knowledge about the system is therefore frequently not available to the maintainers. Often a system's documentation is out of date and/or insufficient, leaving the source code as the only reliable source of information for maintenance programmers. Reverse engineering tools have been developed, as describe later in section 2.2.4, to allow for regaining some of the lost insights of these existing systems. It has been estimated that

the complexity of the existing systems and the lack of documentation and supporting tools result in a maintenance cost amounting to a 60-75% of the total cost of software product during its lifecycle [BOEH81].

Program comprehension is an essential part of software evolution and software maintenance. The importance of program comprehension has been studied in [RUGA95a] [RAJL02]. Software that is not comprehended cannot be changed or maintained. The fields of software documentation, visualization, program design, and so forth, are driven by the need for program comprehension. Program comprehension also provides motivation for program analysis, refactoring, reengineering, and other processes. Programs that are not maintained properly end up aging [GARG98] [HUAN95] and eventually must be retired.

### 2.1.1 Comprehension Gaps

A major part of the software maintenance process is devoted to the comprehension of the system in question. More than 50% of maintenance time is spent on comprehension activities such as reading documentations, browsing codes, understanding the system architecture etc. [FJEL83]. During forward engineering, programmers make a set of decisions, taking into account the restrictions and challenges at the time. Some compromises are made and the product is delivered within schedule with a set of functionalities. However, decisions and compromises made during forward engineering are not specified or directly reflected in the source code and are rarely documented. As a result, a reverse engineer is left with many unanswered questions.

Why was it done this way? Could I have come up with a better design? These sorts of questions are representative for the challenges one faces during program comprehension. Rugaber [RUGA95] identified five comprehension gaps a maintainer typically has to overcome.

**1) Application domain and program**

A program is a solution to a problem in a particular application domain. In order to find a solution one must have the knowledge about the initial problem and thus the domain application. For example, the domain model of an Operating System would contain knowledge of OS components such as memory management and other OS structures. This knowledge often takes the form of specialized schema including design rationalization such as the use of "first come first serve" versus "Round Robin" scheduling. Without the domain knowledge of the OS architecture, one would have a difficult time understanding what the program models.

**2) Physical machines and abstract description**

In order to understand what task or functionality the code is supposed to perform, one must identify the important concepts, a process called abstraction. Yet, a given code section might be part of several abstractions, such abstractions are said to be interleaved [RUGA95b]. This could easily confuse the person reading the program code. There is typically no documentation of the interleaving in the source code.

### 3) Coherent models and incoherent artefacts

It is rare to find programs that have up-to-date documentations. Frequently, programs are upgraded and maintained while the documentation remains unchanged. As a result, the original structure has deteriorated [BELA71] and the program documentation and the program code do not relate to the same information. What is a programmer to believe when the documentation specifies one thing yet the code is clearly performing something else? Is the documentation out of date or is the code not following the original specification /design document, whom to trust? Moreover, as the application undergoes several maintenance iterations, the application might become incoherent within itself. For example, a functionality called "getCustomerNumber" was used in one part of the code as "getting the customer ID" whereas at some other code portion, the code was used as "getting the number of customer count". What is the correct usage of "getCustomerNumber" in the first place?

### 4) Hierarchical programs and associative cognition

Computer programs are highly structured and organized in specific concepts such as functions, expressions and declarations. In order to understand the program, the reverse engineer must be able to build mental abstractions (also called chunks [LUKE80]) from these concepts. A program is understood to the extent that the programmer can correctly build correct high level chunks from the low level concepts present in the program.

**5) Bottom-up program analysis and top-down model synthesis**

There exist two traditional ways to analyze a program. In the first approach the low level concepts are considered as part of a higher construct intended to accomplish a larger purpose. In this case it is called the bottom up approach [SOLO84] [LETO86]. On the other hand, a programmer can have some idea of the overall purpose of the program and how it might be assembled, thus as the program is being studied, concepts are refined into more complete descriptions by adding lower level details. This is the top-down approach [BROO77]. The challenge is that both of these activities need to proceed at the same time in a synchronized fashion [ORNB92].

### 2.1.2 Cognitive Models

To overcome these gaps, different programmers adopt different comprehension strategies, depending on the task at hand (debugging, adding functionalities, refactoring etc.). Researchers have identified several cognitive models used by programmers while performing different comprehension tasks.

**1) Bottom-up**

Programmers read source code and "chunk" low-level software artefacts into more meaningful higher-level abstraction [SHNE80]. These abstractions are then grouped to gain a higher understanding of the program. Programmers using bottom-up strategy gather code statement and controls- flow information to mentally create the application low level structure. These structures are chunked and cross-referenced by

macro-structures to form a program model using application domain knowledge to produce a hierarchy of functional abstractions [PENN87].

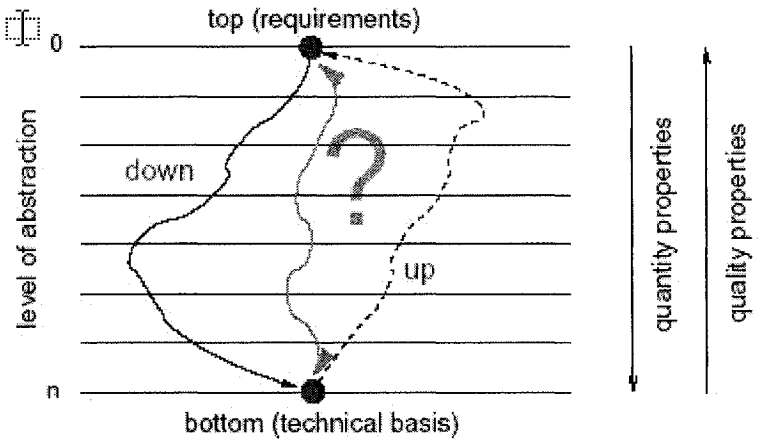


**Figure 1: Bottom-up and Top-down reasoning, the bottom up process uses technical basis "chunk" to derive requirements [PIZK03]**

For example, a programmer will associate a "for loop" with a "sort algorithm function". Then the "sorting algorithm" is linked as part of a higher abstraction such as "sorting a priority task list". A schematic representing the bottom-up approach and the top-down reasoning is presented in Figure 1.

**2) Top-down**

In the top-down approach, programmers reconstruct knowledge from the application domain and map this knowledge back to the source code [BROO83].

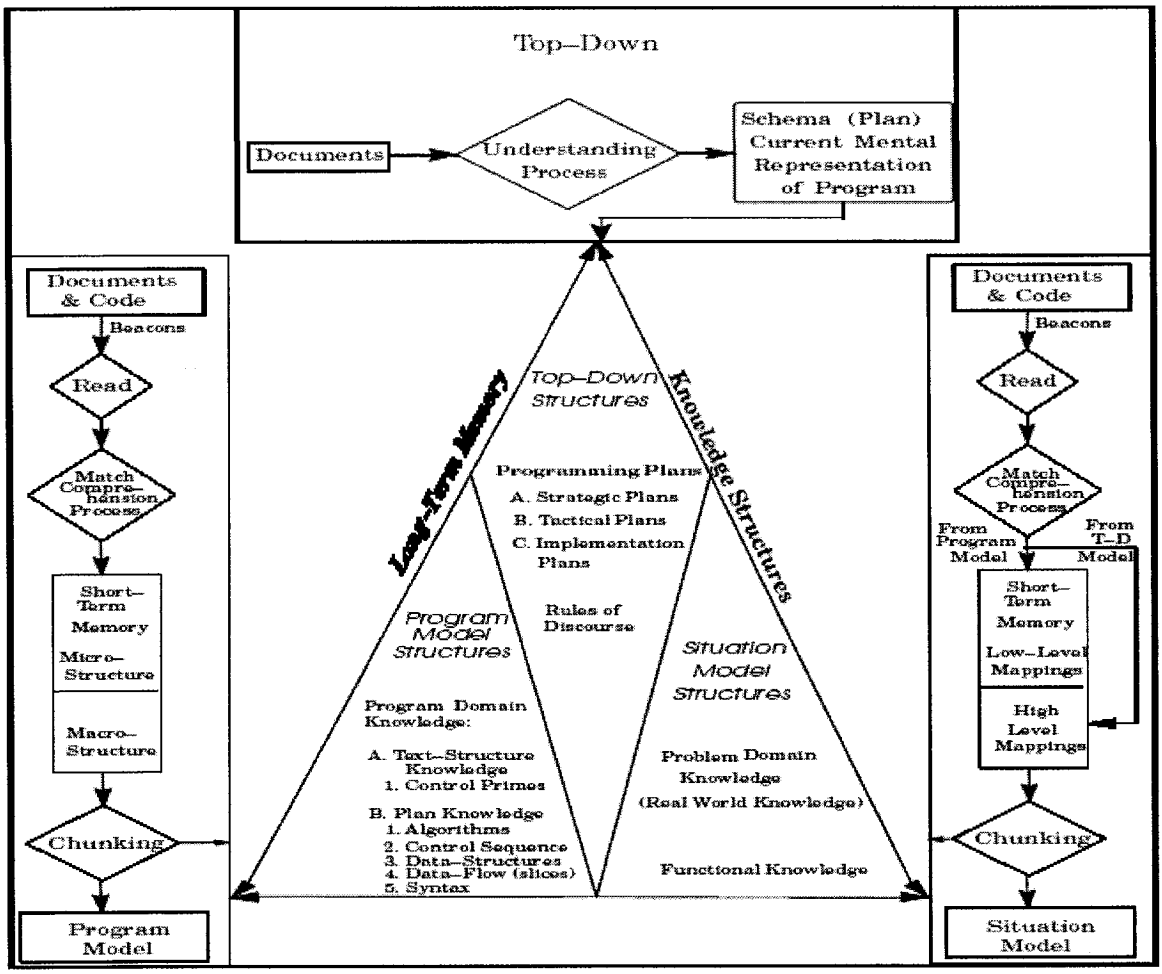


**Figure 2: The cognitive process of the "Top-Down" approach [KEOW00].**

Programmers have a general hypothesis about the program. Verifying or rejecting the hypothesis depends on the presence of beacons (cues) detected by the programmer while exploring the code [PENN87]. Top-down strategy is often used when the programmer is faced with a familiar program type [SOLO84][MAYR98]. Expert programmers recognize program plans and exploit programming conventions (design patterns) during comprehension. The Top-Down represents knowledge about the application domain, see Figure 2.

### 3) Knowledge-based

The assumption for this comprehension model is that programmers are opportunistic and are capable of exploiting either bottom-up or top-down cues [LETO86]. As depicted in Figure 3, this theory has 3 major components;

- Knowledge base: the programmer's application and comprehension expertise.
- Mental model: the programmer's current understanding of the program.
- Assimilation process: how the mental model evolves using the programmer's knowledge as program information is absorbed.

Inquiry steps are keys to the assimilation process. It consists of the programmer asking questions and finding answers as he searches through the code and documentation to verify his answers.

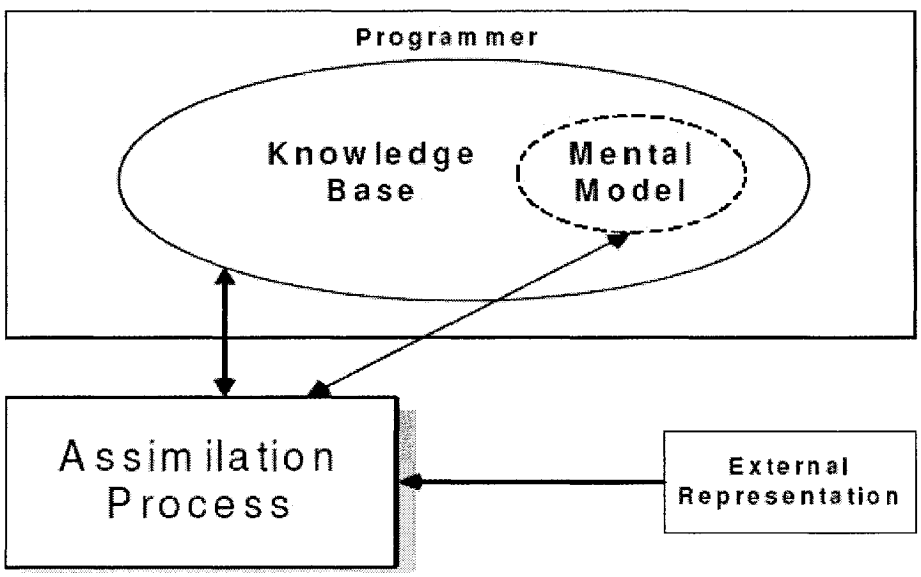


**Figure 3: Using a knowledge base to understand programs [OBRI03].**

**4) Systematic and "as needed"**

For the systematic "as needed" approach, programmers either use a systematic approach, reading the code in details and tracing through all control and data flow, or an "as-needed" approach, focusing only on the code related to the task at hand [LITT86], see Figure 4. However, the systematic approach is less feasible for larger programs while the as-needed approach could generate more mistakes since important interactions might be overlooked. Modifying codes based on an incomplete understanding of the program may be error prone.

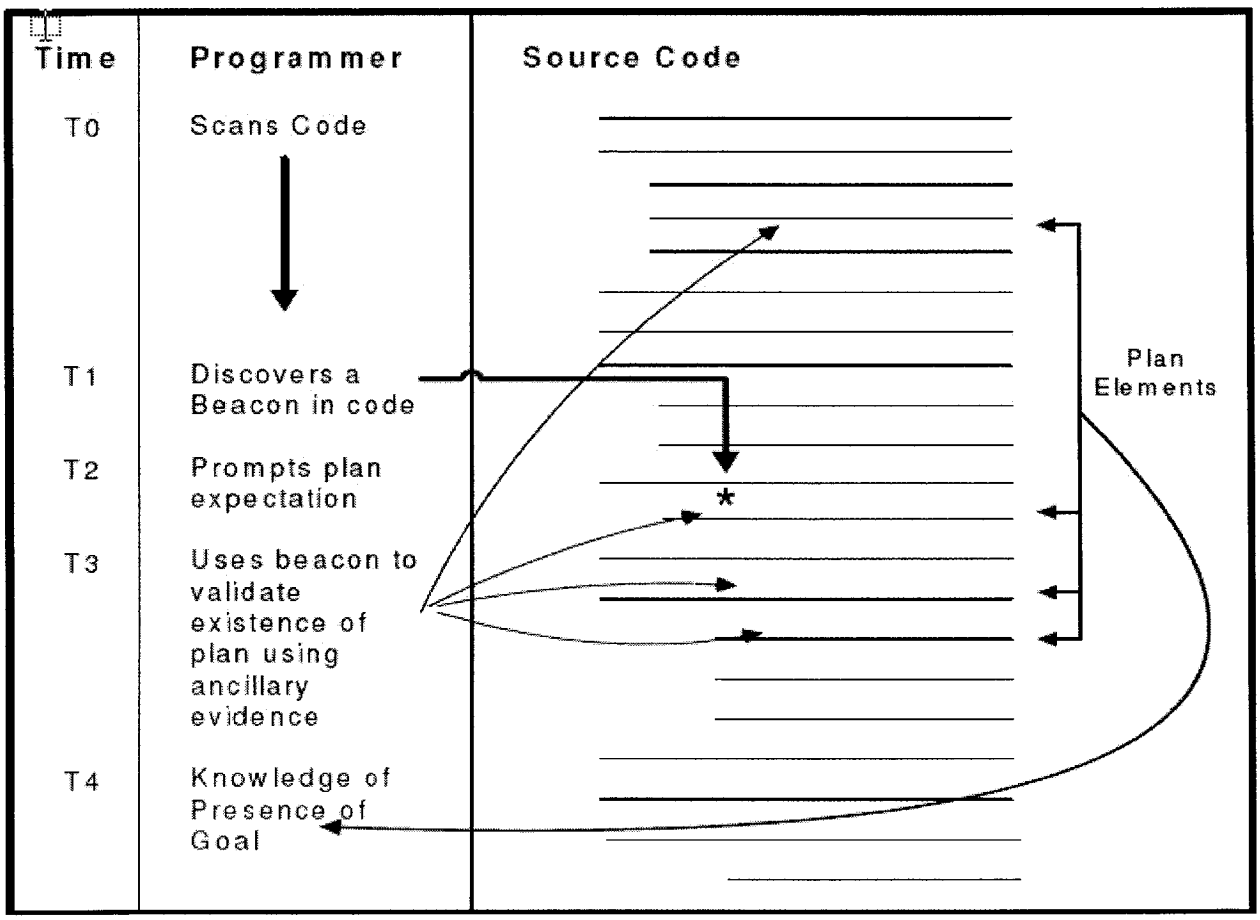


**Figure 4: Opportunistic approach while parsing code [OBRI03]**

### 5) Integrated approaches

This approach is a combination of bottom-up, top-down and knowledge-based comprehension model, integrating them into a single metamodel as proposed by [MAYR95] and demonstrated in Figure 5 [OBRI03]. In practice, programmers freely switch between the three comprehension strategies, adopting the strategy that fits best for the task at hand. At any moment during the comprehension process, any of the above comprehension models can be applied. For example, during the program model construction a programmer recognizes a beacon, indicating a common task such as a sorting algorithm. This leads to the hypothesis that the code sorts something that triggers the use of the top-down model. The programmer has some objective in mind and searches the code for clues to support his expectations or hypothesis. If during the search he finds more unrecognized code, he may jump back to program model building (top-down).

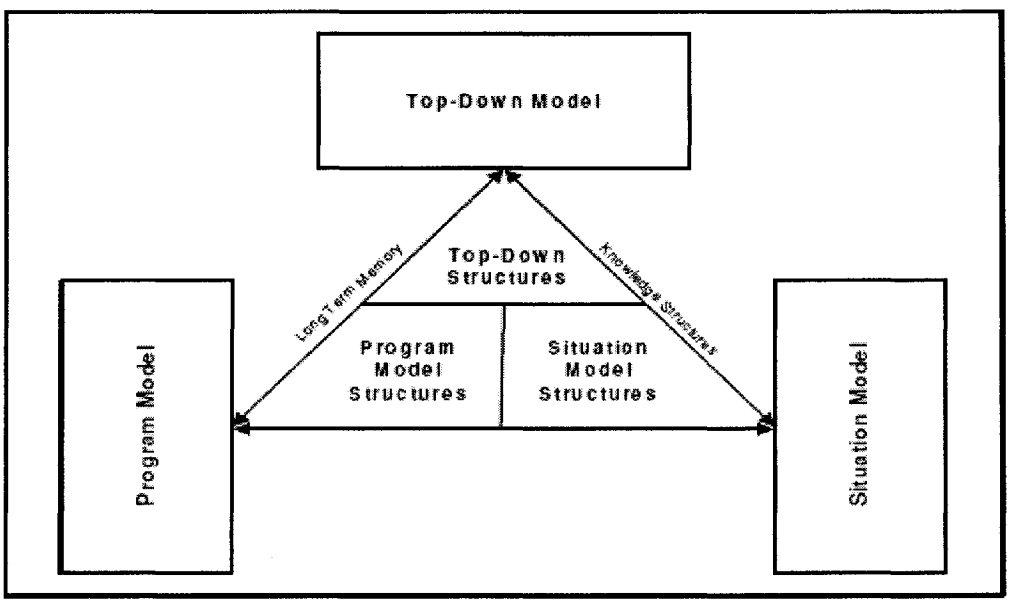


**Figure 5: The integrated approach as per Mayrhauser & Vans metamodel [MAYR94]**

## 2.2 Software Visualization

Software Visualization (SV) is "the use of the crafts of typography, graphic design, animation and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software" [PRIC93b].

When approaching an unknown piece of software for the first time, programmers always find it a challenge to comprehend it unless it is of the most trivial kind. Understanding software is a major challenge within the software development and maintenance domain. Great expense is required to employ people to analyze software for modification, alteration, bug fixing, and maintenance. Certain measures such as meaningful comments in code and up to date documentation can reduce this problem. The recent rise in popularity of object-oriented programming was supposed to be a solution to the problem of software comprehension. Although object orientation (OO) brought a richer model to programming, it is not a total solution to the problem. Current software is too large and too complex for a single human to digest in a reasonable amount of time. Moreover, traditional means of navigating through these large systems (e.g. text editors) have not adapted well to these new challenges.

Comprehension tools can facilitate program comprehension by using visualization to disclose the high level structure of programs. At its best, software visualization should be equivalent to having the original author available to discuss the program structure [KEOW00]. Generally when a new programmer confronts a piece of software and has some confusion over a part of the software, or needs clarification, the

quickest solution would be a consultation with the original author. Original developers have a mental picture or conceptual model of the software that they will transform into program code [JERD94]. The goal of visualisation is to serve as a comprehension tool similar to having the software architect as mentor. Software visualization tools should help the programmer to focus on program sections that is relevant to the programmer's task at hand instead of having to look at the entire application code.

The ultimate aim of visualization is to speed up and improve the production and maintenance of software. Software is very expensive to create and maintain, so obviously the more efficient these processes are, the better [JERD94], [PRIC93a], and [ROMA92].

### 2.2.1 Traditional Visualization Techniques

1. **Line color-coded representation**

Typically, software visualization has mainly been code visualization. Software codes were highlighted in meaningful colors, and then the code lines were reduced in size to allow a more global view of the source code [BALL96].

**Figure 6: Color-coding from Visual Assist [V_ASSIST].**

Figure 6 represents C++ type coloring. Strings, macros, functions, variables and other types are color-coded. This provides a quick recognition of types and facilitates comprehension. Color-coding could represent anything from program functionalities to program slices and even to code version history. However, the drawback manifest itself when there are too many color codes whereas the user is overwhelmed with a multicolour text and coloring just becomes a nuisance to the comprehension process. Color-coding is not scalable and of course it would be meaningless to color-blind people.

## 2. Directed Acyclic Graph

Program executions could be presented and visualized as call trees and call graphs, see Figure 7. As the name implies, these graphs represent what functions are called and who is calling whom. Directed Acyclic Graphs (DAG) are useful for visualizing program behavior and the interaction between codes portions. With a DAG, it becomes easier for programmer to see the code behavior without having to go through a program execution trace. Again the scalability of this technique can be a restricting factor.

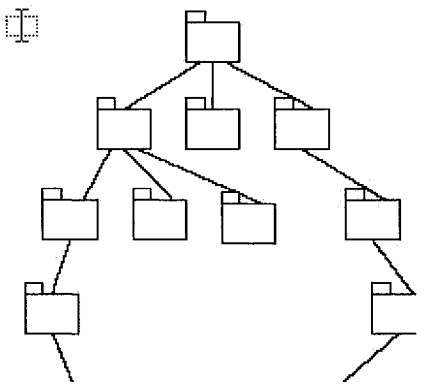

**Figure 7: A tree graph of packages [FOUR04]**

Furthermore, although the call graph does indicate what functions are being called, it fails to demonstrate the calling order and number of times a function is being called. An even more serious problem is dynamic call graph in object-oriented programming. Representing recursive calls within a call graph is a challenge.

### 3. Tree Maps

Tree maps are 2D representation of the program source code. Each rectangle represents a file source code of the program. The rectangles are directly proportional to the file size relative to the entire program.

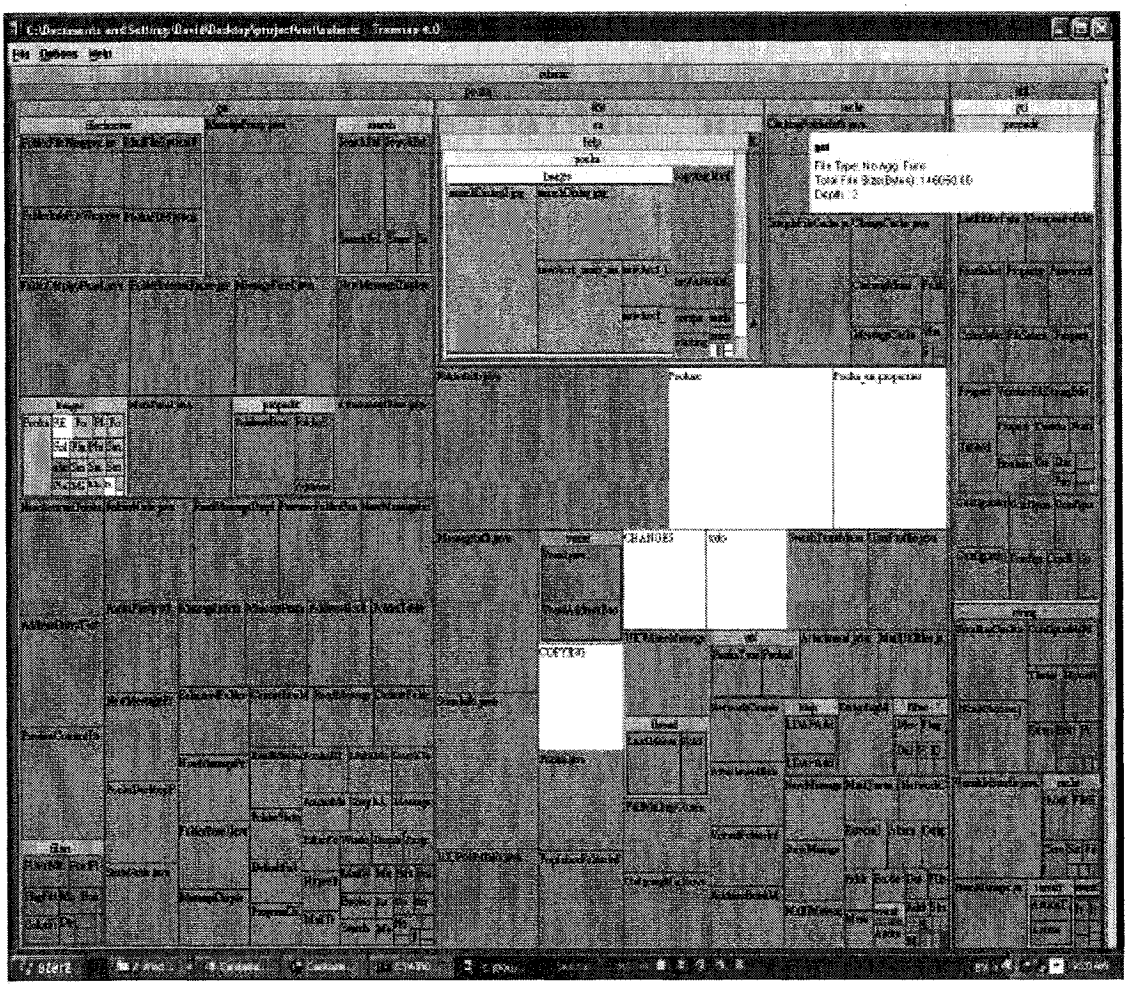

**Figure 8: Tree map of file distribution [FOUR04]**

A quick glance at a tree map can show the programmer the main cores of the program code and what files are more important in size (code volume content) than others. Figure 8 shows a "Tree map" demonstrating the size of each file within the application.

### 4. UML representations

The object-oriented programming has an intuitive nature for modeling the reality. UML class diagram is an intuitive technique to model object-oriented class hierarchy. A class diagram represents objects that the code models and the relational hierarchy amongst them. It helps the user divide the program code into smaller and more manageable code portions where each portion represents an object or concept of the real world.

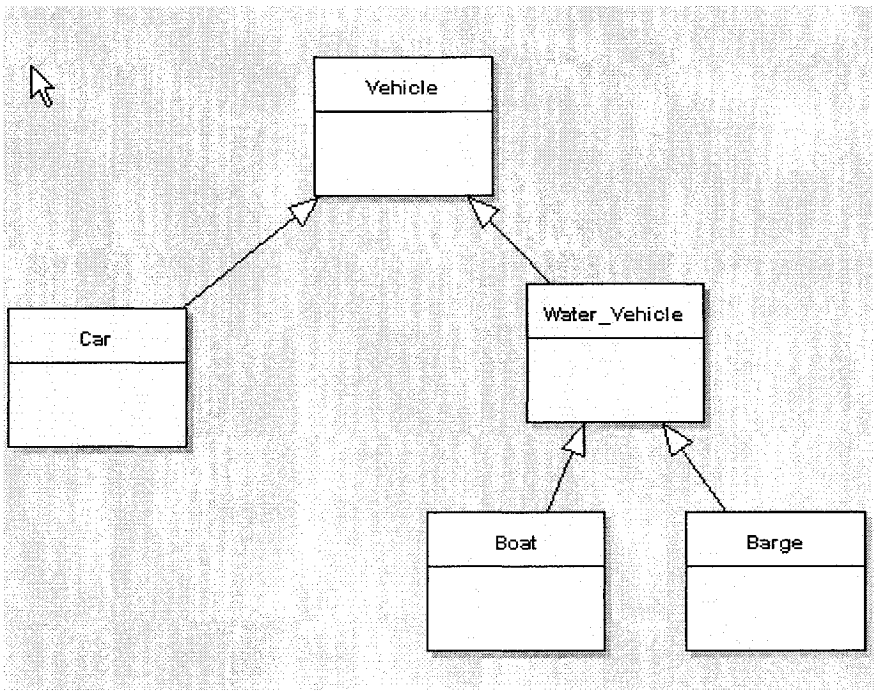


**Figure 9: UML Class diagram**

An UML class diagram helps the programmer to visualize code "chunks" and code "beacons" within the application. Figure 9 shows a typical UML class diagram structure and classes hierarchy. It clearly depicts what real world object the code is modeling.

### 2.2.2 Main Challenges of Software Visualization

Visualization is a major technique in software comprehension. It allows software engineers to create a concrete view out of intangible design ideas. If an image is worth a thousand words then software visualisation (SV) is worth a thousand lines of codes. It is an extra channel of communication among software engineers in addition to words and codes. It allows the comprehension of larger ideas instead of code details. As useful as software visualization might be, the technique still faces major challenges. As listed by [YOUNG96], they are:

1. **Metaphor:** How to represent the different entities of the software?

A large part of the difficulty in providing intuitive visualizations is the lack of physical structure in software. A question arises immediately, "What does the software look like?" According to [BROO87], softwares have no "visualisable" structure despite progress in restricting and simplifying them. They remain inherently "unvisualisable", and thus do not permit the mind to use some of its most powerful conceptual tools. Creating visualizations that are both useful and intuitive is a difficult problem.

2. **Abstraction**: What to present and in how much detail?

Another problem common to many SV systems is *information overload.* A view may seem useful and intuitive for small, simple examples, but may quickly become too cluttered to understand for larger software. Acyclic graphs are a natural representation for many software artefacts. They usually consist of nodes and links carefully arranged by a layout algorithm. The most difficult aspect of showing software through graphs involves the graph layout problem. The nodes and edges must be positioned in an informative layout that clearly shows the underlying graph's structure [BALL96]. Unfortunately, drawing informative graphs is exceedingly difficult for large systems. In the case of acyclic graphs, the numbers of nodes can become overwhelming and the numbers of edges crossing each other can render the graph unreadable.

3. **Navigation**: How to move through the set of visual objects?

When looking at a large set of data, one must be able to locate and focus in the interested portion in a quick and intuitive manner. Navigating through the presented information should not hinder the comprehension process and it should be as transparent as possible. However, large systems require large screen space to display. Where two related artefacts are displayed too far from each other (perhaps even on another screen page) navigating back and forth between pages can hinder the general comprehension process due to a sudden jump from one view to the next and therefore interrupting the context.

4. **Correlation**: How to link the abstractions to the source code and documentation?

After all, software visualization is only a picture of the source code. It remains difficult to show that the visualization represents the actual source code. The abstraction is merely a summary of the source code. Since it is possible to convey all the information of the source code in the abstraction what information should be kept and what should be discarded? Keeping too much information will create an abstraction that is too complex to understand and leaving out too much will create a view that might lose some crucial information from the code.

### 2.2.3 Criteria for evaluating a SV system

Many software visualization tools exist. Some are more efficient than others. How does one evaluate the quality of a software visualization tool? Every traditional visualization technique exposed in the previous section has some drawback. It is difficult to come up with a tool that satisfies all visualization purposes. [STAP99] proposes the following key criteria to evaluate a software visualization system.

- **Usefulness**: The visualization makes the programmer's job easier. A useful visualization should provide information that is not readily available by direct inspection of the source code. That is the purpose of the visualization tool. If a tool cannot justify its usefulness, then it simply cannot justify its existence.
- **Intuitiveness**: The visualization is easy to understand. It should match the programmer's intuition about what the software "looks" like. Although it is accepted that software has neither form nor shape [BROO87], the software engineering community has established some standards in the domain of software

visualization. A class is generally represented as a rectangle whereas a directed arrow represents a relationship. Unless the abstraction attempts to introduce new ideas in the representation, those well-established abstractions should remain unchanged.

- **Scalability**: The visualization works well for large (real world) systems. It should not become less useful or less intuitive as the size of the visualized system increases. Although most tools will serve their purpose in small and medium size program code but as the code volume increase, the visualization becomes too complex and the amount of information presented to the user becomes overwhelming. Facing real world systems that are large and complex the usefulness and the intuitiveness disappear causing failure of the visualization tool. This is by far the greatest challenge of software visualization. Information overload hinders the comprehension capability of the user and will eventually render the visualization totally useless.

### 2.2.4 A Review of Existing Commercial and Research Tools:

There exist several software visualization tools that can provide valuable information to software engineers about program codes and guide these programmers during the comprehension of large software systems. However, these tools also suffer from several weaknesses. The following sections will evaluate a few of these tools and discuss their strengths and weaknesses.

### 2.2.4.1 CC- Rider

A product of Western Wares [http://www.westernwares.com/], *CC-Rider* is a tool to develop, visualize and document software written in the C and C++ programming languages.

The **CC-Rider Visual Browser** provides information on classes, functions, variables, strings, comments, templates, enum values, name spaces, and macros. A *CC-Rider* interface is presented in Figure 10.

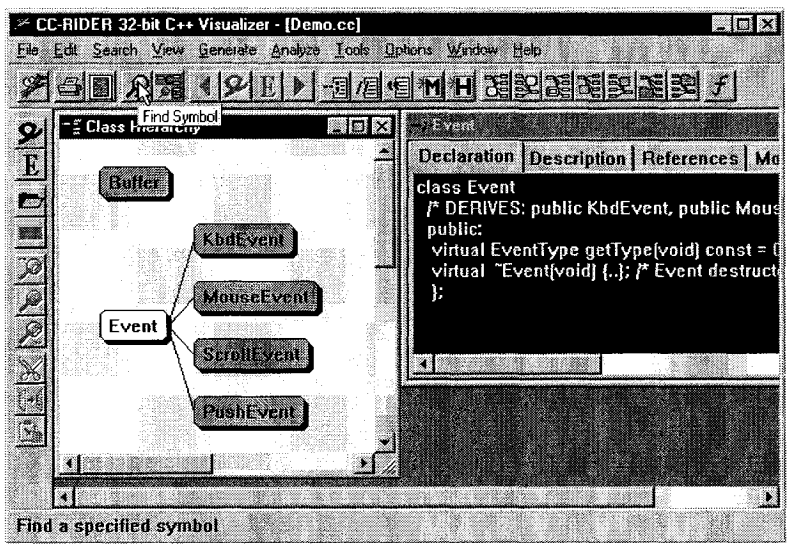


**Figure 10: CC-Rider Interface**

*CC-Rider* also provides additional numerous useful views and filtering techniques such as:

- **Class Hierarchy** represents the class inheritance structure of the program. System's root classes are shown to the left. Derived classes are displayed to the left with connections to the primary parent classes.

- **Class Ancestry** displays a reverse graphical representation of the class inheritance structure of the program. It differs from the Hierarchy View in that individual derived classes are shown on the left, with all their parent classes shown to the left. Branches in this graph indicate multiple inherited classes.
- **Class Nesting** displays a graphical representation of the class nesting structure of the program. These are classes defined within the scope of another class. In this tree, classes with nesting are shown on the left, with nested classes to the right. Double-clicking on any class in the Hierarchy, Ancestry or Nesting windows will bring up details for the class.
- **Call/Caller Trees** display complex relationships between the functions, methods and data in the applications. Function calls and data references are represented as differently shaped nodes in the tree. These trees are useful for examining the structure of C applications, which do not have the object-oriented class relationships.
- **File Relationship Views** display graphical representations of the files number included by other source files in the system.
- **Symbol Find** is a list box showing various alphabetic ranges of raw symbol names.
- **Program Statistics** display the symbols stored in the CC-RIDER Database by the source code Analyzer.
- **Documentation generator** generates documentation for the projects in various formats (RTF, html and winHelp)

**CC-Rider strength**

- Understanding and Navigating Code: Code reading and navigation is made easier with *CC-Rider*. The visualization could clarify future development, accelerates maintenance and quickly bring new team members up-to-speed.
- Code Documentation: Whether for maintenance, commencing a new project or accommodating a new team member, *CC-Rider* enables a quick understanding of the source code.
- Team Collaboration: Programming teams can share code documentation over the Internet with HTML documented version of the project thus enhances communication among team members.

**CC-Rider weaknesses**

- Scalability: *CC-Rider* performance and usefulness diminishes as the program and diagrams get larger causing the programmer to suffer information overload.
- Domain knowledge: while the diagrams are useful to describe the current status of the source code, it does not explain why it has been written that way.

#### 2.2.4.2 Source Insight

A project oriented program code editor and code browser, with built-in analysis for C/C++, C#, and Java programs, *Source Insight* (http://www.sourcedyn.com/index.html) parses source code and maintains its own database of symbolic information dynamically while the programmer works, and presents contextual information automatically. *Source Insight* also display reference trees, class

inheritance diagrams, and call trees. The vendor claims that *Source Insight* was designed for large, demanding, real world programming projects, see Figure 11.

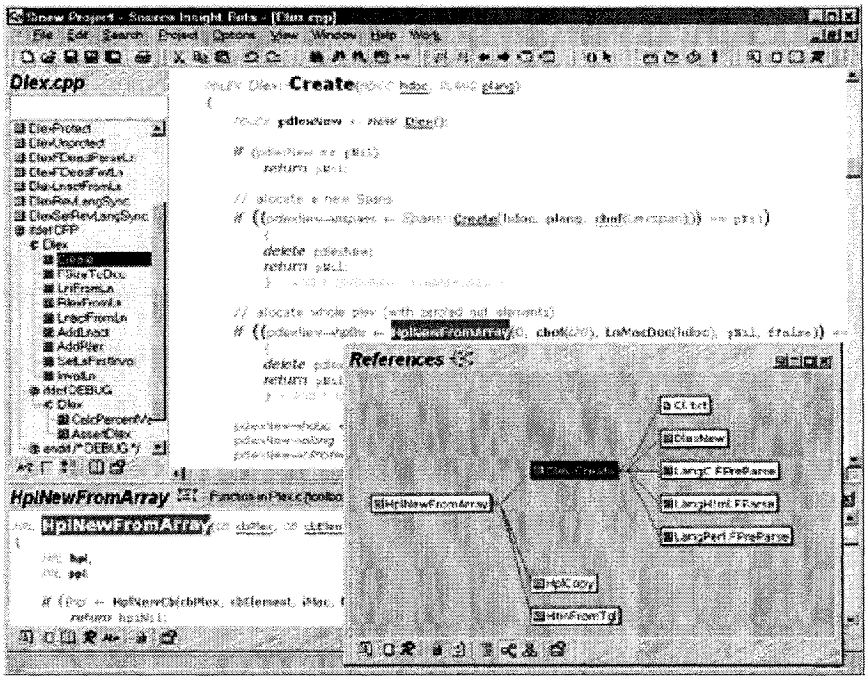


**Figure 11: Source Insight interface**

**Source Insight main features**

- Always Up-To-Date information: *Source Insight* automatically builds and maintains its own symbol database of functions, methods, global variables, structures, classes, and other types of symbols defined in the project source files. The symbol database provides browsing feature, without having to compile the project or having to depend on the compiler to provide browser files.
- Call Graphs and Class Tree Diagrams: The Relation Window shows relationships between symbols. Programmers can view class hierarchies, call trees and reference trees.

- Syntax Formatting: Information while programmers read their code. For example, references to local variables can look different from references to global variables, references to functions and references to C function-like macros can also appear different. With Syntax Formatting, it becomes obvious what an identifier refers to, or if it is misspelled.
- Context-Sensitive Smart Rename: Source Insight's indexes allow programmers to rename variables, functions, and other identifiers in one step. Source Insight's context-sensitive can rename local scope variables and global or class scope identifiers.
- Team Programming Support: Changes made by any member of a programming team are reflected automatically since the entire code base is scanned and resynchronized as needed. Programmers can instantly jump to the definition or usages of any symbol, and can access modules and other symbols without having to know what directory, machine, or file they are in.

**Other feature includes**

- Mixed Language Editing.
- Keyword Searches like an Internet Search on Code Base.
- Symbolic Auto-Completion.
- Access to All Symbols and Files
- Hyper Source Links to Link Compiler Errors and Search Results
- Project-Wide Search and Replace
- Project Window with Multiple Views

**Source Insight weaknesses**

Although *Source Insight* is a powerful source code editor and browser, it remains a source code editor helper. It neither provides great help in understanding the architecture of the program nor does it help in understanding the application's concepts. *Source Insight* is useful for the development process but it will offer little efficiency for comprehension and maintenance. Even though the vendor claims that *Source Insight* is useful for large software, the reading of source codes (even facilitated by all the features) can quickly become overwhelming to anyone as the software grows in size. *Source Insight* will need other comprehension means to override its weaknesses.

### 2.2.4.3 Understand for C++

*Understand for C++* (http://www.scitools.com/ucpp.html) is a reverse engineering, documentation and metrics tool for C and C++ source code, see Figure 12. It offers code navigation using a cross reference, a syntax colorizing "smart" editor, and a

variety of graphical reverse engineering views. *Understand for C++* is an interactive development environment (IDE) designed to help maintain and understand large amounts of legacy or newly created C and C++ source code.

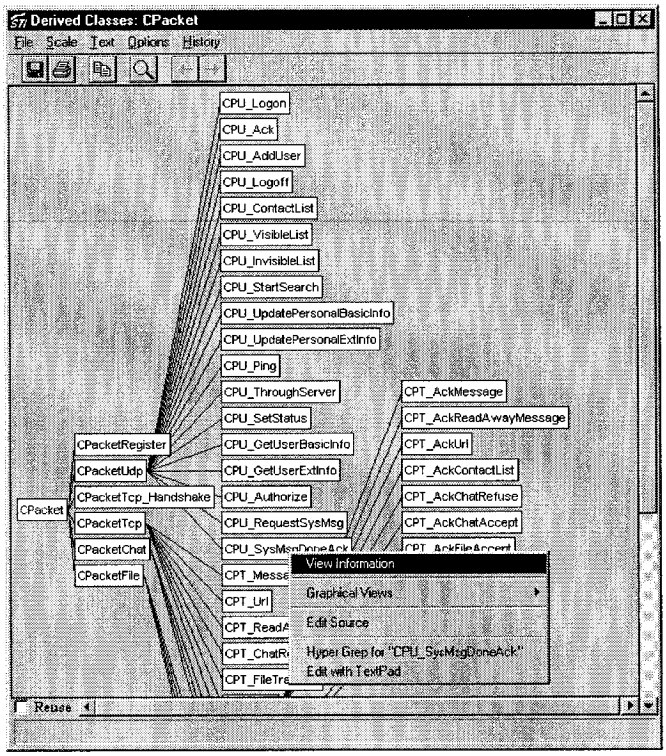


**Figure 12: Understand for C++ class hierarchy**

*Understand for C++* offers a variety of reverse engineering diagrams such as:

- **Class Inheritance Diagrams** reverse engineer class inheritance relationships. The **Derived Classes Tree** documents how a C++ class was built (inheritance). The **Base Class** tree shows what other classes depend on a class (inherit from it).
- **Invocation Trees** document a calling hierarchy between C and C++ functions from the top to the bottom (or from the bottom to the top). There are two types of trees. The **Call Tree (Invocation)**, indicates what is accomplished by a function

call (or causes to be called). The **CallBy Tree**, reveals who calls a function, permitting the programmer to easily trace the impact of a change to that function.

- **Include Trees** show what depends on a given C or C++ header (.h) file (include file dependency) as well as what header files a given section of code depends on.
- **Declaration Diagrams** are available for any declared item that has structural relationships with other items. It also documents the first level of uses and dependencies.
- **Class Diagram s**hows what a members a class offers as well as where it derives from, and what derives from it.
- **Include File s**hows what macros, functions, "typedef" and classes are provided by the included file. Also shown are the other files that include this file (dependencies) and what this file includes (depends on)
- **Function s**hows parameters and return type. Also shown are the files that this function depends on (calls) and those that depend on this function (called by).
- **Data Member Diagrams** are available for any declared class, type or "struct" that has data members. The diagram shows all components and how they are being built. The diagram follows and unfolds the types used to build members.

**Understand for C++ strengths**

It provides a great variety of views and diagrams for programmers. Given the many views options, the programmer can choose the appropriate diagram that would best fit his comprehension task. It is a tool meant for reverse engineers and provides a wide range of comprehension tools to understand the source code.

**Understand for C++ weaknesses**

Since these diagrams are in 2D the difficulty of the comprehension task increases exponentially as the program becomes larger. Diagrams quickly become overwhelmingly large and too much information is presented in a single view. Multiple view windows are good filtering techniques but it will become confusing to manage too many opened windows as the program increases in size.

### 2.2.4.4 SNiFF+

*SNiFF+* (http://www.windriver.com/products/development_tools/ide/sniff_plus/) is an Integrated Development Environment (IDE) for Unix or Windows application developers who are working with large volumes of code (generally 100 KLOC to 5MLOC) using C, C++, Java, or ADA or a mixture of any of those languages. It is a source code analysis environment for software developers and teams who work with large amount of application code. The SNiFF+ tool promotes engineering productivity and code quality by providing a comprehensive set of code visualization and navigation tools that enable development teams to organize and manage code, see figure 13. It also support reverse engineering, configuration management, workspaces and build management.

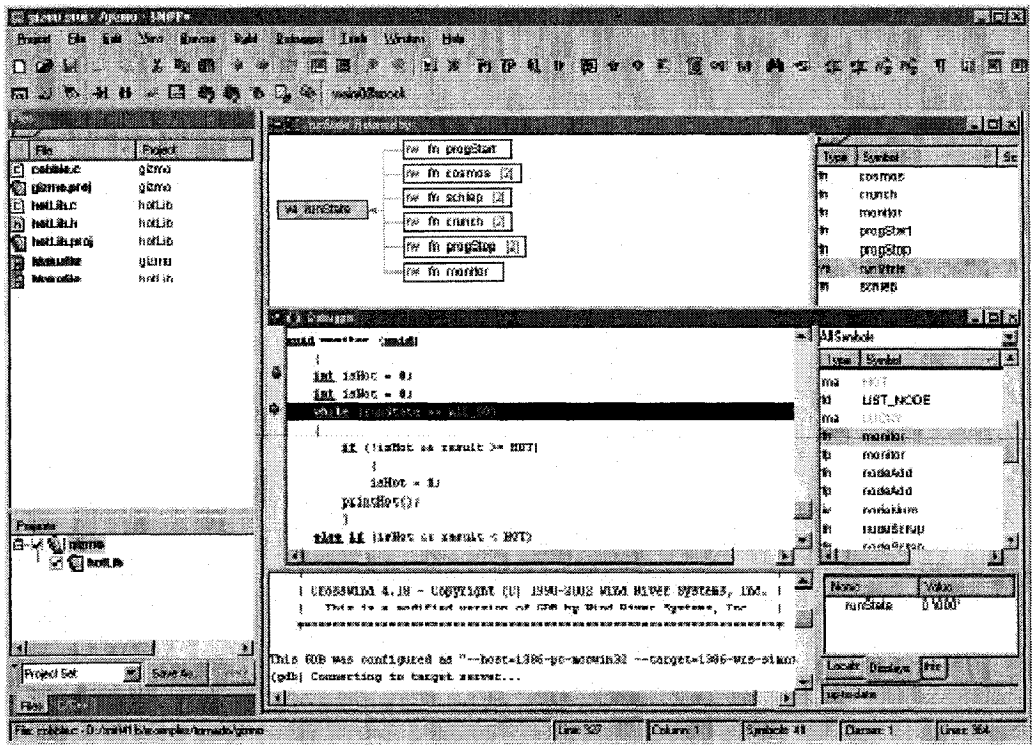

**Figure 13: Sniff++ interface**

*SNiFF+* is a powerful source code browser and a great IDE. It provides more functionalities than traditional IDEs such as Eclipse, Visual Studios or JBuilder. However it remains a development tool more than a reverse engineering tool. Its functionalities promote quick understanding of the source code but provide little insight about the program architecture. It helps in browsing the code and with finding out "what" is in there but it does not provide "why" it is there and "why" it is done that way. For that reason, it can help new developers to understand the current code and continue to add on features but it will be little help when the developer needs to maintain the currently existing features.

#### 2.2.4.5 Imagix 4D

*Imagix 4D* is a comprehensive program understanding tool, it enables programmers to check or study software on any level, from the high level architecture down to the details of its build, class, and functional dependencies. Programmers can visually explore aspects of their software such as control structures, data usage, and inheritance, see Figure 14.

**Imagix 4D Features**

- **Build Dependencies:** This view indicates the interrelations among files, showing which files are required to build other files. Some of these dependencies result from build rules in "makefiles" while others are the result of include statements in the source code.
- **UML Class Diagram:** uses the format from the Unified Modeling Language to depict what UML terms the collaborations, associations and relationships inherent in the software.
- **UML File Diagram:** shows the "include" relationship and the member-to-member relationships (associations). Programmers are able to follow a symbol and its dependencies across files.
- **Function Pointers:** tracks assignments from one function pointer to another and recognizes function pointers or functions passed as parameters.

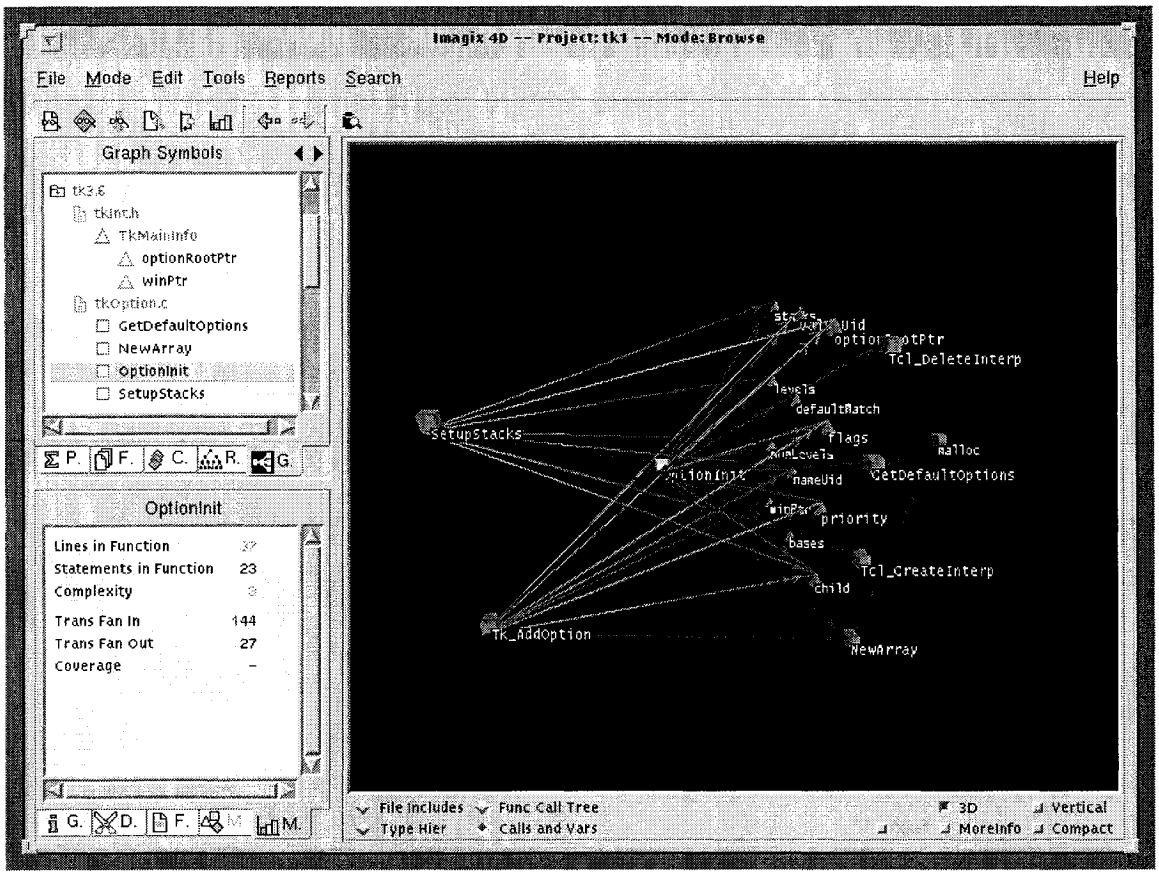


**Figure 14: Imagix 4D interface**

- **Control Flow Mode:** Through the Control Flow graph, programmers can visualise the complex dependencies and understand the critical areas of the code.
- **3D Graph:** All graphs can be view in 3D layouts
- **2D Graph:** All graphs can also be view in 2D layouts which are more intuitive than 3D

### Imagix 4D strength

Providing a large variety of potentially useful views it is one of the rare tools that allow the user to visualize the global program architecture. Without having to browse

though the source code, the user can have a general idea of the program structure with a simple glance at the class diagram.

**Imagix 4D weaknesses**

Tool's usability diminishes as the program size increases. Diagrams quickly become overwhelmingly complex and the user is presented with too much information. Furthermore, the use of colors quickly becomes a nuisance and it becomes difficult to understand their metaphor.

### 2.2.4.6 A Tool Summary

| Tool | Strength | Weakness |
|---|---|---|
| **CC-Rider** | Easy code navigation<br>Documentation generation | Scalability |
| **Source Insight** | Good code editor<br>Good code browser | Only good for code view, no insight on program architecture |
| **Understand for C+** | Variety of views<br>Variety of filtering techniques | Scalability<br>Only support 2D |
| **Sniff++** | Good IDE<br>Good code browser | No support for program architecture comprehension. |
| **Imagix 4D** | Provide visual architecture<br>Supports 3D views | Scalability<br>Color coding becomes confusing |

In general, only some tools support a 3D environment and they all suffer a scalability issue at one point or another. It is also evident that the main vocation of these tools are for the forward engineering process and provide little values for reverse engineering.

## 2.3 3D versus 2D Visualization

Traditional 2D visualization quickly decreases in effectiveness as the systems to be visualized become larger. This decrease in effectiveness is mainly due to the information overload and the limited navigation provided within typical 2D visualization approaches. The information presented creates a significant comprehension burden for the programmer. The comprehension of 2D diagrams is restricted by the resolution limit of the visual medium (usually the 2D computer screen) and the limit of the user's cognitive and perceptual capabilities.

With the development of graphic techniques and the dramatically reduced price of 3D hardware, 3D graphic is gaining more and more attention and becoming an active visualization area. By analyzing some properties of 3D modeling, one can see how the third dimension can be applied to alleviate some problems associated with traditional 2D software visualization [NGAN02].

### 2.3.1 Advantages of 3D visualization

The third dimension can address some of the drawbacks that can generally hinder traditional 2D visualization techniques [KEOW00].

1. **3D makes more effective use of available screen space.**

3D visualization can use the additional dimension to encode additional knowledge due to the larger working volume. The additional dimension can be used to handle either larger information volume compared to 2D visualization (enhancing information density), or to provide the ability to visualize additional types of information to enrich the visuals.

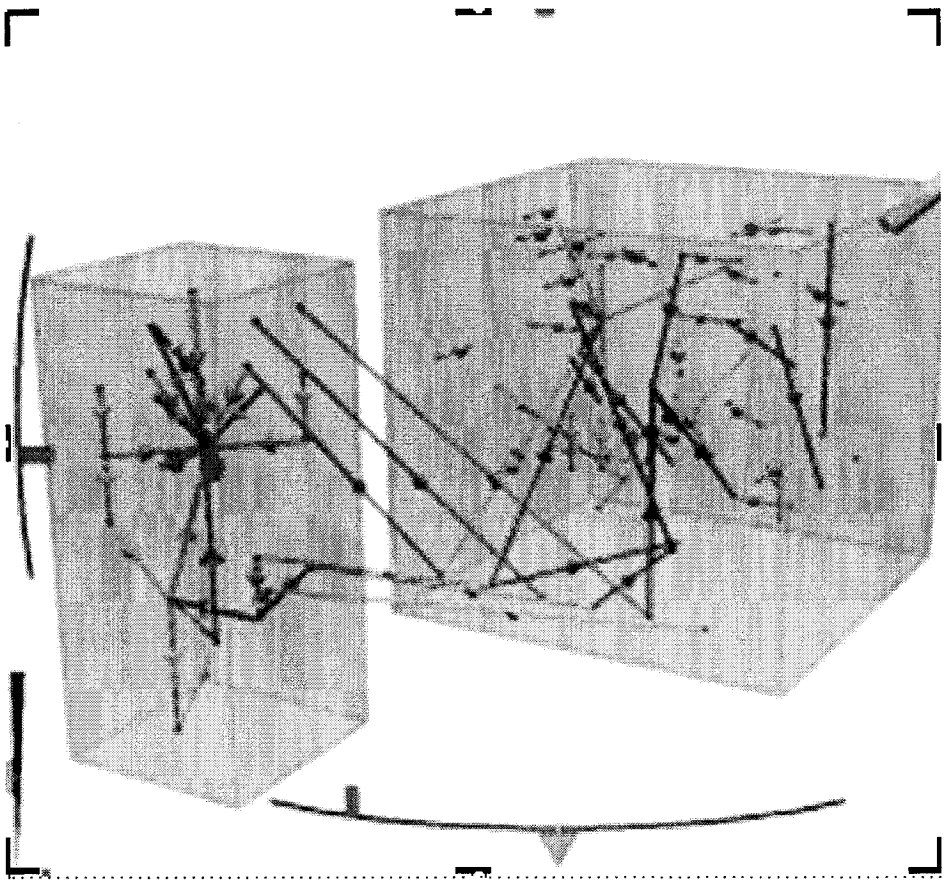

**Figure 15: Two distinct grouping of objects [PITT98].**

In step with cognitive science and human perceptual system, 3D software visualization can provide the intuitive exploration and interaction of the system. Mapping artefacts into a 3D space allows users to identify common shapes or common configuration that may become apparent, and which could be then related to design features in the code.

As an example, Figure 15 shows two distinct groups of objects. They represent two different modules of an application. It is clear that the box shape objects represent the module containing other elements.

2. **Enhancing information density.**

The average distance between entities in 3D space is less than in an equivalent 2D space [NEIL98]. 2D diagrams must spread out, whereas 3D has the potential to be more compact [DWYE01]. If one takes the world map as an example (Figure 16), when it is wrapped around a sphere Australia and Hawaii are spatially close to each other. They are at opposite sides on a 2D map, a user would be forced to scroll for some time to move from one country to the other, see Figure 16. By using 3D compact space, the average distance between countries is reduced. 3D visualization browsing should theoretically be faster and simpler due to the density of entities.

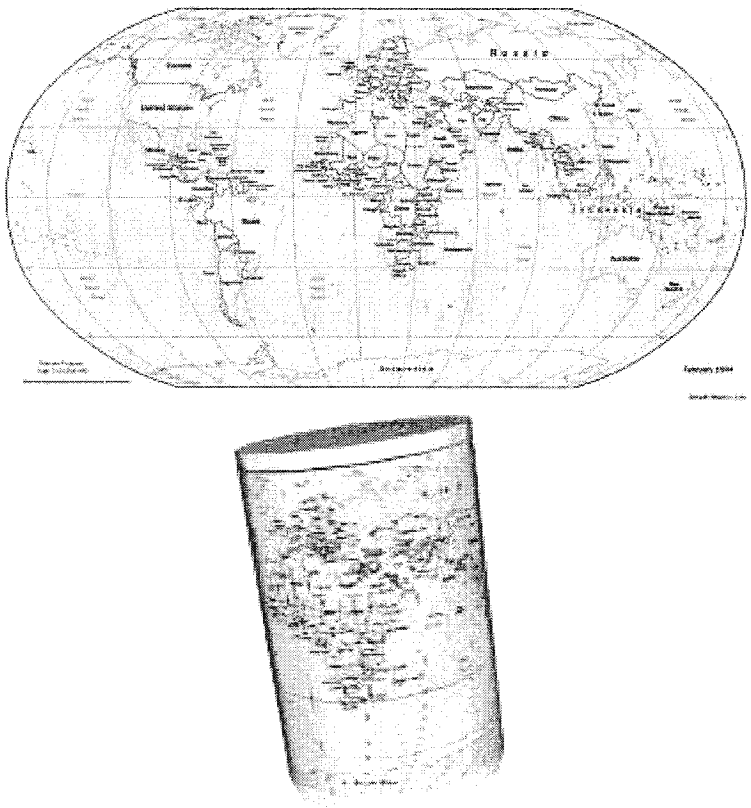

**Figure 16: World map analogy, entities are placed closer to each other in 3D.**

3. **Reduction of visual clutter**

Links and relationships among objects in 3D space are less likely to cross (or come close) than in 2D diagrams. In a complex 2D diagram, in which entities may be linked to other entities to show relationships, it is often impossible to avoid the crossing of links. If this happens too often diagrams can become very confusing and virtually unreadable. 3D diagrams help to avoid this potential problem. There is no single plane that links are restricted to, they can travel in any direction. It is rarely necessary for links to cross in a 3D diagram, allowing for a reduction of the necessary link crossings in a 3D diagram [DWYE01], see Figure 17. Combining with the ability to view the diagram

from various angles without having to recreate the view (only available in 3D), the user can choose the viewpoint that provides the least links crossing.

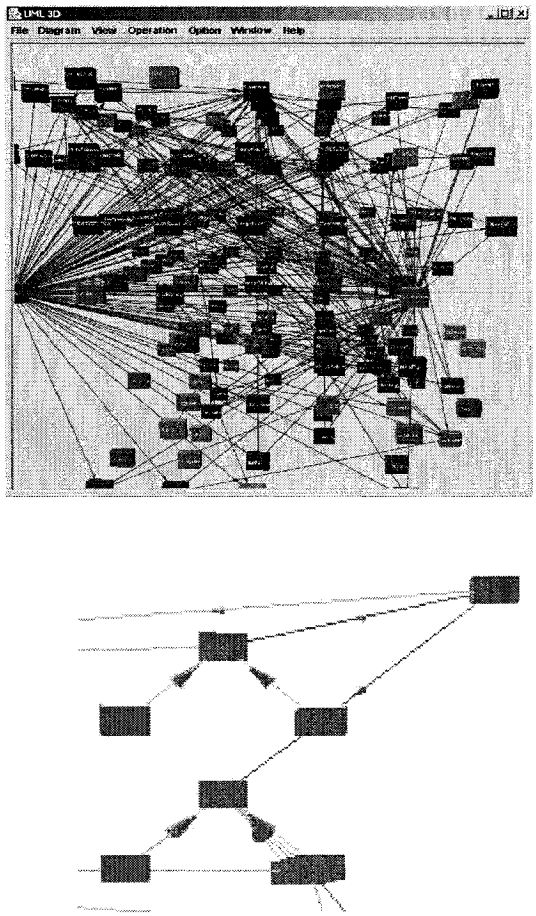

**Figure 17: 3D reduces edge crosscutting, edges are not only restricted to a plane.**

4. **Increase in representational range.**

3D characteristics of entities can represent properties of underlying objects. For instance a sphere may represent class X, a pyramid class Y, etc. Although a similar regime is possible in 2D, 3D allows a much greater scope, since the potential complexity of shapes can be much greater. Transparencies can be used to express the notion of belonging or to generate a relevant meaningful space that can be easily be interpreted by

the user, see Figure 18. Less important 3D objects can also be drawn in wire frames, thus highlighting full 3D shaded objects. Wire frames indicate the object's presence but objects that are fully 3D rendered will be the focal points. The greater the number of means of representation available, the more meaningful and "information dense" the diagram becomes.

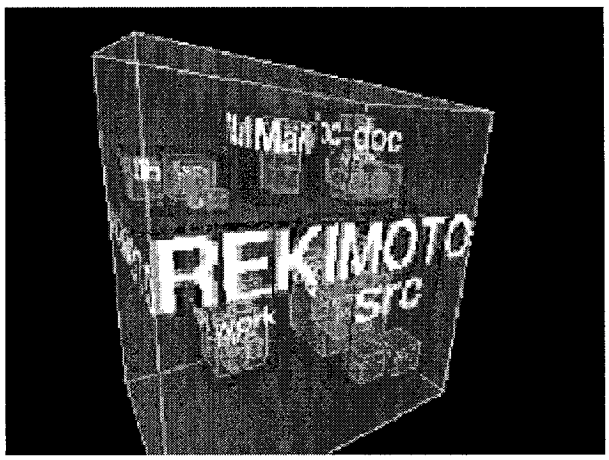


**Figure 18: 3D transparency, one object is contained inside another.**

5. **The positions of objects in space may have relevance to their properties.**

For example, objects below object X have property P, objects behind have property K, etc. In two-dimensional diagrams, this is limited to above, below, left and right. Three-dimensional view adds the notion of depth, allowing more information density, see Figure 19. One can now connect meaning to an entity being behind another, or in front of it. The figure below shows that the notion of depth is easier to represent in 3D than with 2D counterpart.

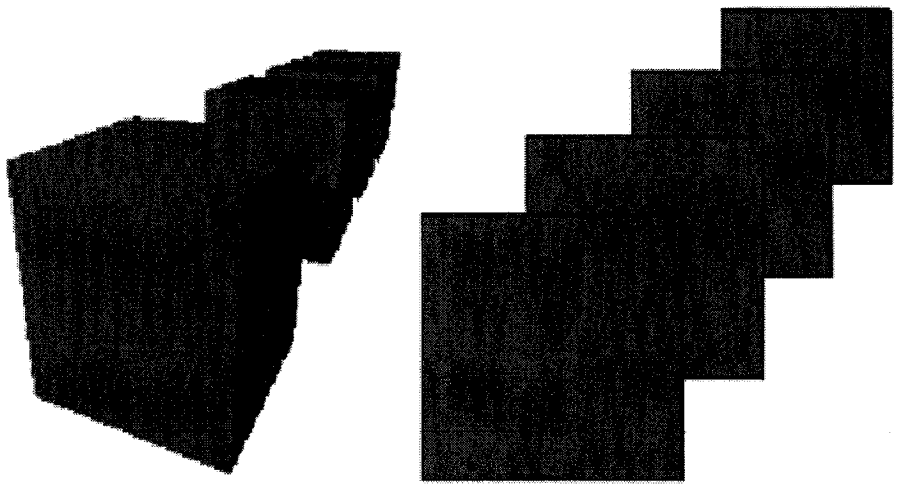

**Figure 19: Depth is easier to represent in 3D than 2D.**

6. **Intuitive Navigation.**

3D offers more interesting options as far as actual viewing is concerned. Whereas a user can scroll around a 2D diagram, they may wish to fly around a 3D view, or walk, or simply stay put and rotate the world around them. When a model is built, the most appropriate means of navigation will have to be considered. For small diagrams, the best solution may be to have the model surround the user, and allow them to study the entire world simply by spinning it around on some axes. For a large diagram there may be tiers of information, the top level may contain different information relative to the second level. Therefore it is feasible that the user may walk around the levels, and fly between them [MALE01].

7. **3D can give a sense of reality.**

3D visualization may impart a sense of "being there" to a user. If users feel that they are traversing a virtual world, they are more likely to become immersed in the world, and be more receptive to details portrayed in the world [MALE01].

### 2.3.2 Challenges facing 3D visualization

Although 3D visualisation provides many advantages, it is not the "Holy Grail" of software visualisation. Along with its promising advantages, it also brings its own drawbacks and challenges. Following are issues that arise with of 3D visualisation:

- **The frame rate** (the rate at which a browser displaying a virtual world can update its screen) must be considered as a potential disadvantage if it is not realistically high. Low frame rate will cause non-fluid screen updates when the user rotates, zooms or navigates around in the 3D space. Reasons for a low frame rate are predominantly technological, due to limitation of adequate hardware support for 3D rendering. If a diagram is unpleasant for users to navigate then they are unlikely to gain anything from it. They are also unlikely to want to use it again. Fortunately, 3D is now quite feasible on the average desktop. With advancement of computer CPU speed and more powerful graphic cards, this issue will be of lesser importance in the future.

- **Disorientation** is another potential shortcoming of 3D. A user may get lost within a diagram, by facing an inappropriate direction, or by moving too close to some

object or too far from the model in general. Disorientation problems are preventable, however, with sensible model design. A user can be prevented from getting too far away from a model by encasing it in walls. Sign posts can also be positioned to give users an idea of where they are. Other more complex help aids may also be devised. For instance, one may imagine programmed help agents, which appear to inform the user of their current location. Similar problems also exist in two dimensions. These issues are being addressed by such techniques as fish eye views.

- **Spatial Navigation** in 2D-graphs visualizations is limited to translation in x and y coordinates, scaling about a point of interest, and rotation about a point orthogonal to the screen. However, rotations are not commonly used. In 3D space users are exposed to much more freedom of movement. If not used properly, navigating in 3D space could result in the programmer being totally lost in the environment and this could hinder the comprehension process. Navigation and orientations are problems that arise hand in hand. With the third dimension available, one must also consider the "world up" and the travel direction. One cannot successfully navigate through the world without proper orientation.

- **Text labels pose a problem in 3D**. In a 2D diagram, text labels cause no problem. They are always readable since they are always viewed from the "front". It is assumed that labels are readable no matter where a user should scroll too. However in 3D there are some important considerations. Should labels

always face the user? Should they scale as the user moves towards or away from them? Should they appear only when the user comes within proximity? Label density is also an issue. If text is too densely packed it may obscure objects in the background or make the entire display confusing, see Figure 20.

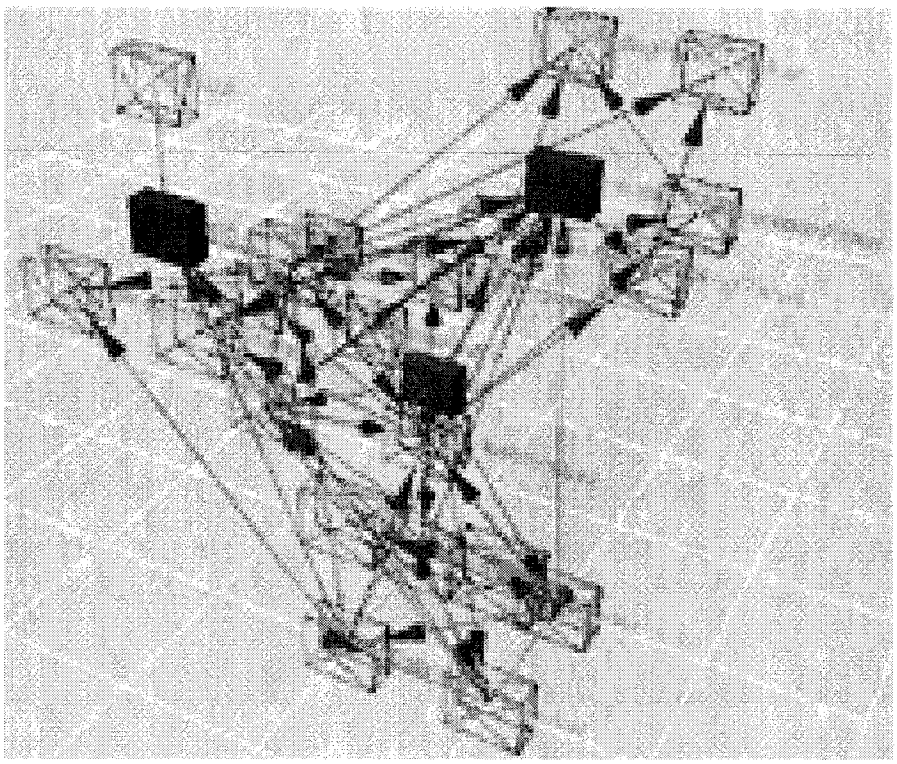

**Figure 20: 3D texts are difficult to read.**

Text labels in 3D also have the disadvantage of being geometrically complex, which can lead to a reduction in frame rate. For this reason we must try to reduce the use of text whenever possible. Another important consideration is that the viewing tool should avoid overwhelming the user with unnecessary data causing information overload. The user should never be faced with an overly complex screen brimming with information.

- **Information overload** can result in diagrams that are unreadable. Utilizing a third dimension, viewing tools are able to display more information on the computer screen.

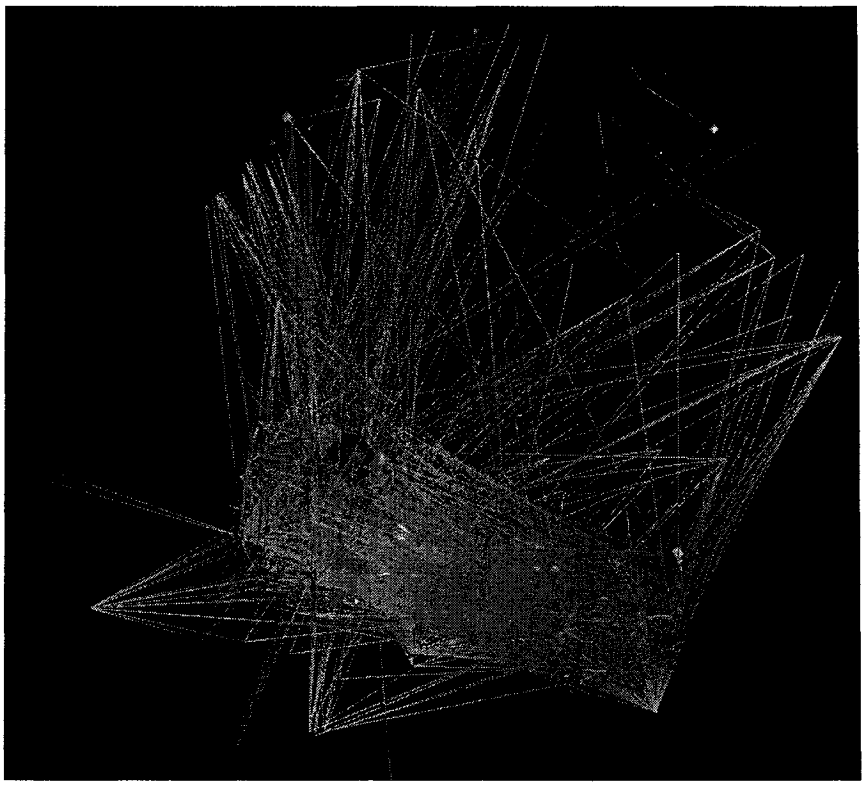

**Figure 21: A system of over 1200 classes [LEWE01].**

From figure 21, which shows a system with over 1200 classes, it becomes obvious that there is too much information condensed into a limited visual space causing the visualisation to be difficult to understand. Thus more information might be presented to the user within a given (limited) computer screen space than using a 2D representation. However, the resulting view can become too complex or cluttered for the user to understand easily. 3D visualization can present in theory more information than 2D but if that information is not

presented in a fashion that is useful to the user, it will only hinder the comprehension process. The key to solve this information overload challenge is to apply "filtering techniques". The user should have the option to specify what information to view and what information should be filtered out, therefore allowing for a reduction of the information to be displayed.

- **Scalability** is still an issue for 3D visualization. Even though 3D representations are more scalable than their 2D counterpart, scalability remains an important issue. As the program grows bigger, more items must be shown to describe its system. As a result, 3D visualisation is again faced with the problem of information overload. Scalability and information overload are usually problems that come hand in hand. New filtering techniques can be applied to address the scalability issue, by only displaying artefacts that provide relevant information and by filtering out the unnecessarily artefacts.

### 2.3.3 Potential solutions to 3D visualization challenges.

There are some potential solutions to the 3D visualization challenges.

(1) *Navigation*: Intuitive ways must be found for navigation through the 3D world to explore the information presented. The information might be dense but with efficient navigation users may find what they need quickly and thus minimizing the effect of information complexity.

(2) *Viewing technique*: the main view is broken to different specific subviews where each of them can be assigned to convey a different type of information.

(3) *Better filtering and analysis techniques*: exclusion of unwanted information, reduction of information overload and improved scalability. Divide and conquer is the solution for large and complex tasks.

Presented below are potential techniques to address these 3D visualization challenges.

**1) Orientation and Navigation in 3D scenes**

Navigation through 2D visualization is generally trivial and simplistic. The user is exposed to generally accepted navigation process such as scrolling with scrollbars, dragging the 2D scene or opening a new view perspective. There are a number of differences between the visual structure of 2D and 3D interfaces. Probably the greatest difference is that 3D interfaces do not offer a unique, comprehensive view in the most situations. Many view angles are available but only a few are useful for comprehension. The user needs to move through the scene to perform the given tasks. 3D interfaces generally offer an approach based on exploration rather than on synthesis, that is the reason why orientation and navigation are two key issues of 3D interfaces [PITT98].

- **Environment orientation**: Orientation is the first requirement for navigation. One cannot know what direction to go without being properly situated within the environment. There are a number of elements people use to orient themselves

within a 3D space. Directional signs are generally used to communicate space location and directions. When looking at a map of a shopping center or a large building, the user can generally locate a marker showing "You are here" to identify his current location. Also the map is usually oriented with an arrow indication the direction "North". Those are the cues used in most 2D maps to help the users to locate and orient themselves. In a 3D world, the view can use an animated object to identify the current location and use the standard x/y/z axis to orient the user.

- **Navigation input:** Once the user has successfully situated and oriented himself he may attempt to navigate through the 3D scene. This is not a simple task since the 3D scene is not similar to the real world that people are used to. Computer visualisation is limited to the input devices that are currently available (mouse, keyboard, game-pad, trackballs etc.) Using a single one of these devices will not be sufficient. The key is to combine these inputs, a multimodal approach, and to have one device complements the weakness of the other [PITT98].

**2) Different viewing techniques**

Most systems for visualizing large information structures use 2D graphics to view networks of nodes and arcs that represent data. To understand large structures, it is often necessary to show both small-scale and large-scale structures. This is usually referred as the problem of focus and context. It is important to provide information about the large-scale structure of the graph, while at the same time allowing users to drill down to an

arbitrary level of detail. There are four techniques to solve the focus-context problem: multiple windows, distortion, rapid zooming, and elision.

- **Multiple windows:** It is common, especially in mapping systems, to have one window that shows an overview and several others that show expanded details [BEDE94]. In Pad++ [BEDE94], multiple windows are called portals and each portal can be individually zoomed. One potential problem with multiple window techniques is that the details are disconnected from the overview and therefore from the overall context. It might be difficult to see where the zoomed details belong to in the overview window.

- **Distortion techniques:** A number of techniques have been developed that spatially distort a graph. Distortion gives more room to the designated points of interest and decreases the space given to less interesting objects. Some techniques such as the hyperbolic lens have been designed to work with a single focus [LAMP95]. Others allow for multiple foci to be simultaneously expanded [NOIK94]. Many of these methods use simple algebraic functions to distort the graph based on the distance from each focus. An alternative method called "intelligent zooming" [SCHA96] [DEER92] uses techniques of graph layout to dynamically resize and reposition parts of a graph based on selected points of interest. The basic concept of every distortion techniques is to spatially expand what is currently interesting and collapse of what is not, thus providing both focus and context.

- **Rapid zooming techniques:** In rapid zooming techniques a large information landscape is provided, but only a small interested part of it is visible through the viewing window at any instant. The user is given the ability to rapidly zoom in to and out of points of interest, which means that although focus and context are not simultaneously available, the user can rapidly and smoothly move from focus to context and back. This may allow the user to cognitively integrate the information. There are a number of 3D techniques that could be referred to as "zooming". These include changing the camera focal length, moving the camera toward an object, and up-scaling an object. In all cases, part of the scene occupies a larger area of the viewing window. In [MACK90] a rapid navigation technique for 3D scenes was presented, referred to as POI navigation. This method moves a user towards a point of interest that has been selected on the surface of some object. At the same time, the viewpoint of the user is brought to a point that is perpendicular to the surface.

- **Elision techniques:** Elision is a technique where parts of the structure are hidden until they are needed. It is achieved by collapsing a sub-graph into a single node. In the intelligent zoom system [BART94], as a node is opened, it expands to reveal its contents while simultaneously adjusting the entire graph to make more space for the already expanded nodes. Any number of nodes can be expanded while the graph is continuously adjusted, creating more space for objects of interest and correspondingly reducing the size of other parts of the graph.

Therefore rescaling and elision work hand in hand. In elision methods, eliding blocks of structure into a single more compact representation provides context. This kind of technique works well in nested graph structures because entire sub-graphs can be collapsed into representative nodes.

**3) Analysis and filtering support**

Filtering is an important intermediate step in reducing the information volume that has to be displayed [SHNE92]. Two major categories of filtering techniques, namely structural and semantic filtering, can be distinguished. The most commonly used structural filtering techniques in software visualization are based on creating hierarchical views of a software system. Information reduction by structural filtering techniques is one approach to improve the scalability and applicability of visualization techniques. However, without a meaningful interpretation, these filtering techniques might sill not be sufficient in interpreting the systems. Source code analysis techniques can be applied to filter and group the analyzed source code to provide for more meaningful interpretation of the visual representations. One well-known approach to source code analysis is program slicing [GUPT97, HARM01, HORW90, KORE88, WEIS82].

- **Program slicing:** "Slicing" is a program analysis and reverse engineering technique that reduces a program to those statements that are relevant for a particular computation. A slice provides the answer to the question "What program statements potentially affect the value of variable *v* at statement *s*?" It was observed [WEIS84] that programmers have some abstractions about the

program in mind during debugging. The process of debugging consists of following dependencies from the erroneous statement *s* back to the influencing parts of the program. These parts may influence *s* either because they decide whether *s* is executed at all (*control dependence*) or because they define a variable that is used by *s* (*data dependence*). A program "slicer" can be used to automatically compute and visualize the slice of the program with regard to the statement *s* and the variables used or defined at *s*. It allows the programmer to focus his attention on statements that are part of the slice. Additionally, the programmer sees any statements that are not part of the slice and can reduce the importance an attention he might pay to them. Program slicing can be used to assist the programmer in many tedious and error prone tasks, such as debugging, program integration, software maintenance, testing, and software quality assurance. Several variants of program slicing have been proposed for these purposes, including static slicing, dynamic slicing, backward slicing, forward slicing, chopping, interface slicing, etc. [GUPT97, HARM01, HORW90]

- **UML and design patterns:** The Unified Modeling Language (UML) is a general-purpose modeling language for specifying, constructing, visualizing and documenting artefacts of software systems [BOOC99]. It provides a collection of notations to capture different aspects of the system under development. A simple glance at the "class diagram" shows the programmer what class exists in the system and the relation between these classes. The "sequence diagram" shows the run time behaviour of the system and shows how classes interact with each

other. Looking at the diagram makes is easier than going through files and lines of codes in order to understand the program.

Design patterns [GAMM95] have become increasingly popular among software developers since the early 1990s. They help developers communicate architectural knowledge, help people learn a new design paradigm, and help new developers ignore traps and pitfalls that have traditionally been learned only by costly experience. Design patterns are usually modeled and documented in the UML [BOOC99]. However, UML does not keep track of pattern-related information when a design pattern is applied or composed. The elements in the model, such as classes, operations, and attributes, in each design pattern usually play certain roles that are manifested by their names. The application of a design pattern may change the names of its classes, operations, and attributes to the terms in the application domain. Thus, the role information of the pattern is lost. Without this information it is no longer obvious which model elements participate in this pattern so it is hard for a designer to identify design patterns in software system designs [VLIS98] [DONG03].

The omission of design patterns in the visual representations can cause several problems. Without the visualization of design patters, software developers can only communicate at the class level instead of the pattern level. Since they do not have access to pattern-related information in system designs, the benefits of design patterns are compromised. Firstly, designers can no longer communicate with each other in terms of the design patterns they use and the design decisions and tradeoffs associated with the use of the patterns. Secondly, each pattern

typically documents some ways for future evolutions, which are buried in system designs. Secondly it may require considerable efforts to identify design patterns manually within an existing software system design [KELL99].

Design patterns not only provide solutions to a recurring problem, but they also convey the rationale behind their solution, i.e., not only "what", but also "why" [BECK94]. Existing work on design pattern recovery typically focuses on the use of static information to recover design patterns. There exists a smaller body of work that also integrates dynamic source code information. Heuzeroth *et al* [HEUZ03] argue that a static analysis is often not sufficient for pattern recovery. They introduce dynamic analysis techniques to detect design patterns in legacy code. Reverse engineering focuses on creating "representations of a system in another form at a higher level of abstraction" [BASS02]. Reverse engineering tools that support design pattern views of the overall system structure can allow for some additional reasoning about "why" certain implementation/design rationales might have been chosen in the original implementation.

- **Grouping and Layout:** Analysis, 3D visualization and filtering techniques on their own are only steps towards improving software visualization. They are typically not sufficient to close the conceptual gap between representations created during the traditional forward engineering and those created as part of reverse engineering. The success of a visualization technique depends also on its ability to visually organize and decompose a system. It has been shown [BASS02,

FAVR01] that grouping, clustering and layout can improve readability by supporting representations that closely related to the mental model programmer forms of a system during typical comprehension tasks [MALE01, MAYR98, STOR99].

Many factors influence the original layout decision of the visual notation used while modeling or designing a software system, e.g., functional, domain knowledge, architectural aspects or just pure personal preferences that cannot easily, if at all, be extracted from the source code. Traditional layout algorithms focus mostly on improving the readability by reducing the number of edge crossings in the diagrams [DWYE01]. Other issues, like mapping the visual representation to the programmer's mental model are ignored. The major objective of a good layout and/or grouping algorithm has to be the ability to recreate a representation that (1) closely matches the perception a user has of the system, either based on previous involvement with the system or based on existing domain knowledge, and (2) provide meaningful interpretations (views) of the system which might not correspond to a typical metric based layout/grouping approach. However, it must be noted that recreating a graph layout that matches the original layout created by the designer/developer is a near impossible task.

Typical clustering approaches in software visualization are based on a hierarchical clustering or metric based approaches. For large systems, the problem of minimizing the metric becomes a greater challenge [BECK94]. Not only does the complexity increase when trying to find an optimum solution, it is often impossible to minimize the overlapping and crossing of edges. This quickly leads

to visuals that are very difficult to comprehend. The major challenge of visual clustering and grouping is one of reducing visual aliases. Depending on the number of objects, clustering priority must be shared between user interpretations and clearer visualization. For the forward engineering of new systems, software visualization has been established as the tool of choice to control their complexity. Typically, UML diagrams are used as the graphical view to represent static and dynamic aspects of a software system [DONG03, DWYE01]. However, the combination of applying both hierarchical and non-hierarchical relations poses a special challenge to a graph layout tool.

### 2.3.3 Key issues for the Implementation of a Software Visualization tool.

Not all software visualization tools have been effective in their application. For software visualization to be effective, they must satisfy several identified key issues [LANZ02].

- **The overall architecture:** how the entire software visualization tool is structured. It needs a clear separation between the core, the visualization engine and the metamodel. Flexibility is the key, as the software will likely evolve as development progresses and requirement changes. The metamodel may need to be refined and the visualization techniques may be required to change. An architecture that cannot cope with changes is doomed and the tool may someday need a complete rewrite. Since software visualization is still a field under

research, changes are inevitable, the tool must be flexible to accommodate any new visualization idea or techniques

- **The internal architecture**: the design of the core domain model. This should be simple at first sight but the domain model must be easily extendable. Adding new functionalities and requirements should be simple and should not compromise the integrity of the internal architecture. It is important since research requires exploration of different avenues and the internal architecture should be solid to support those experiments.
- **The visualization engine**: software visualization tools that have special needs and commercial libraries do not provide the degree of freedom needed. On the other hand, writing a complete visualization library from scratch is cumbersome and should not be a burden to the tool provider. A compromise must be made that would provide enough flexibility without having to put most of the effort in writing the visualization engine.
- **The metamodel**: the way data is collected and stored. Although not directly related to software visualization, it is an important issue. The data should be separated from the visualization engine. In this way, data is provided from an external tool. As long as the input data is of the same format, the SV tool can use it.
- **The interactive facilities**: the direct manipulation provided to the user. Although hard to validate, it is the aspect that requires the most work and ultimately dictates the tool's usability and success. The user not only wants to look at the software, most of the times he also wants to interact with the

visualization. Static visualization seldom offers satisfaction to the user. The direct manipulation must also be intuitive and efficient. Slow response manipulation will hinder the ability of the user to understand the software and reduce his will to use the tool again.

# Chapter 3 Contribution

## 3.1 Objective and specific goals

This research focuses on two major areas: (1) Extension of the traditional 2D UML representation into the 3D space. (2) Combination of 3D UML with source code analysis techniques and 3D navigation techniques to facilitate the exploration of the virtual world. The resulting contribution of this research also consists of two parts. Firstly, source code analysis is used to extract design patterns that had been used during forward engineering and to create a layout that would make them easier to recognize and to comprehend. Secondly, a prototype tool entitled **3D-UMLViz** implements a 3D navigation system that reduces the disorientation while navigating the 3D world.

## 3.2 Hypothesis

Source code views are primarily based on some diagrammatic notations that have evolved from the early days of computing [KNIG99]. For large and complex software systems, the comprehension of such diagrammatic depictions is limited by the resolution limits of the visual medium (2D computer screen) and the limits of a user's cognitive and perceptual capacities. These limitations are results of the following problems:

- *Scalability*: For large systems, the resulting visual representations tend to be cluttered, often resulting in information overload problems.
- *Layout:* The awkward and sometimes arbitrary layout techniques tend to focus on minimizing line crossing and often lead to the obstruction of important software aspects such as patterns and other types of relationships.
- *Non-intuitive navigation:* Typical navigation includes pan-zoom and overlapping of multiple windows, which confuses the user when he must browse amongst multiple views.

Visualization mapping techniques such as fisheye-views [FURN86], perspective information walls [MACK91] and hyperbolic trees [LAMP95] offer some solutions for focus versus detail. Although useful to assist the user in not getting lost in the visual space they are limited in their scalability for large software.

It is expected that 3D UML combined with design pattern recovery and filtering techniques can enhance the comprehensibility of the existing metaphors and help close the conceptual gap between forward and reverse engineering. On the other hand it is recognized that 3D visual representation, source code analysis techniques and existing metaphors will not provide any improvement in the comprehensibility of visual representations. Closing the conceptual gap between forward and reverse engineering will require further effort.

In the last several years, three-dimensional software visualization has been recommended in consideration of its advantages over 2D diagrams [JING03] [KNIG99], many of which have been described in detail in section 2.3.1. It is believed that

representing an UML diagram in 3D can significantly reduce the size of the diagram. An extra dimension is available and new viewing metaphors can be divided to convey additional concepts or information that cannot otherwise be expressed in a simple 2D representation

Design patterns [GAMM95] consist of sets of classes that represent a design concept. They are high-level design elements that address recurring problems in object oriented design. A design pattern not only provides a solution to a recurring problem, but also conveys the rationale behind the solution, i.e., not only "what", but also "why" [BECK94].

A significant effort has been devoted to recovering design patterns by focusing on the use of static information such as source code while some works also include tracing dynamic code execution to detect behavioural patterns. Source code analysis can be used to detect design patterns and group the participating classes in a 3D representation to simplify diagram pattern matching. Furthermore, those design patterns have specific layouts that are familiar to most programmers as suggested in [BOOC99]. It is perceived that the comprehension process can be improved if the classes were grouped in the same or similar layout. This is due to several factors. One can understand the classes' roles due to their roles in the design pattern (which is known and documented in the existing literature) [BOOC99], therefore the comprehension process can be moved to a higher level of abstraction, enabling visual pattern matching, and their interpretations. This type of pattern matching is similar to recognition a stellar constellation (pattern) as a whole instead of individual stars.

With regards to information overload, numerous techniques have been explored to overcome this problem. Filtering is an avenue to explore in trying to tackle this issue. Filtering and slicing techniques were discussed in section 2.3.3 as potential solution to 3D visualisation challenges.

The forgoing analysis justifies the effort to introduce 3D visualization regardless of the overload issue.

## 3.3 Research goals

Commercial reverse engineering tools such as "UML studio" (www.pragsoft.com/products.html) and Rational Rose (www-306.ibm.com/software/rational/) simply detect and reverse engineer program classes and connect them with relationship such as dependency and inheritance. They provide neither insight into the program source code nor information on the general structure. The recovered 2D UML diagrams scatter classes over the 2D display without logical positioning or minimizing edge crossings. The diagram sometimes span over several computer screens. Overlapping of relation edges makes the diagram almost unreadable, especially for large systems. The badly presented information becomes overwhelming for the user who cannot concentrate on the relevant information. Such diagrams are of little used for program comprehension.

In an attempt to overcome these difficulties the present research is directed at the following goals:

- Extending the UML representation to 3D space to take advantage of programmers' familiarity with UML and make use of the extra screen space and more advanced rendering techniques that are available for 3D representations.
- Combining 3D visualization techniques with source code analysis to provide programmers with more insights about the software design than mere traditional UML diagrams. The prototype tool **3D-UMLVis** detects design patterns and adds this extra knowledge in its UML representation.
- Making design patterns easier for the user to locate by using layout and grouping that closely matches the 2D layouts introduced by the GoF [BOOC99] to reduce the conceptual gap between forward and reverse engineering. The focus will be on the visualization of static design patterns defined by the GoF.
- Proposing advanced navigation and filtering techniques to reduce disorientation while navigating the 3D virtual world.

## 3.4 Workflow for 3D-UMLVis within CONCEPT

Source code parsing, code analysis and design pattern recognition have sustained research interest for many years [SHAU98, NOBL03]. The recovery of design patterns and their application in comprehending the program during maintenance have recently gained momentum. The CONCEPT project [RILL02] develops a program comprehension environment that supports various levels of abstraction. The CONCEPT

environment is built around a layered architecture supporting a variety of analysis and visualization plug-ins, it also includes a Java3D based prototype implementation of a 3D UML diagram. An architectural representation of CONCEPT is presented in Figure 22.

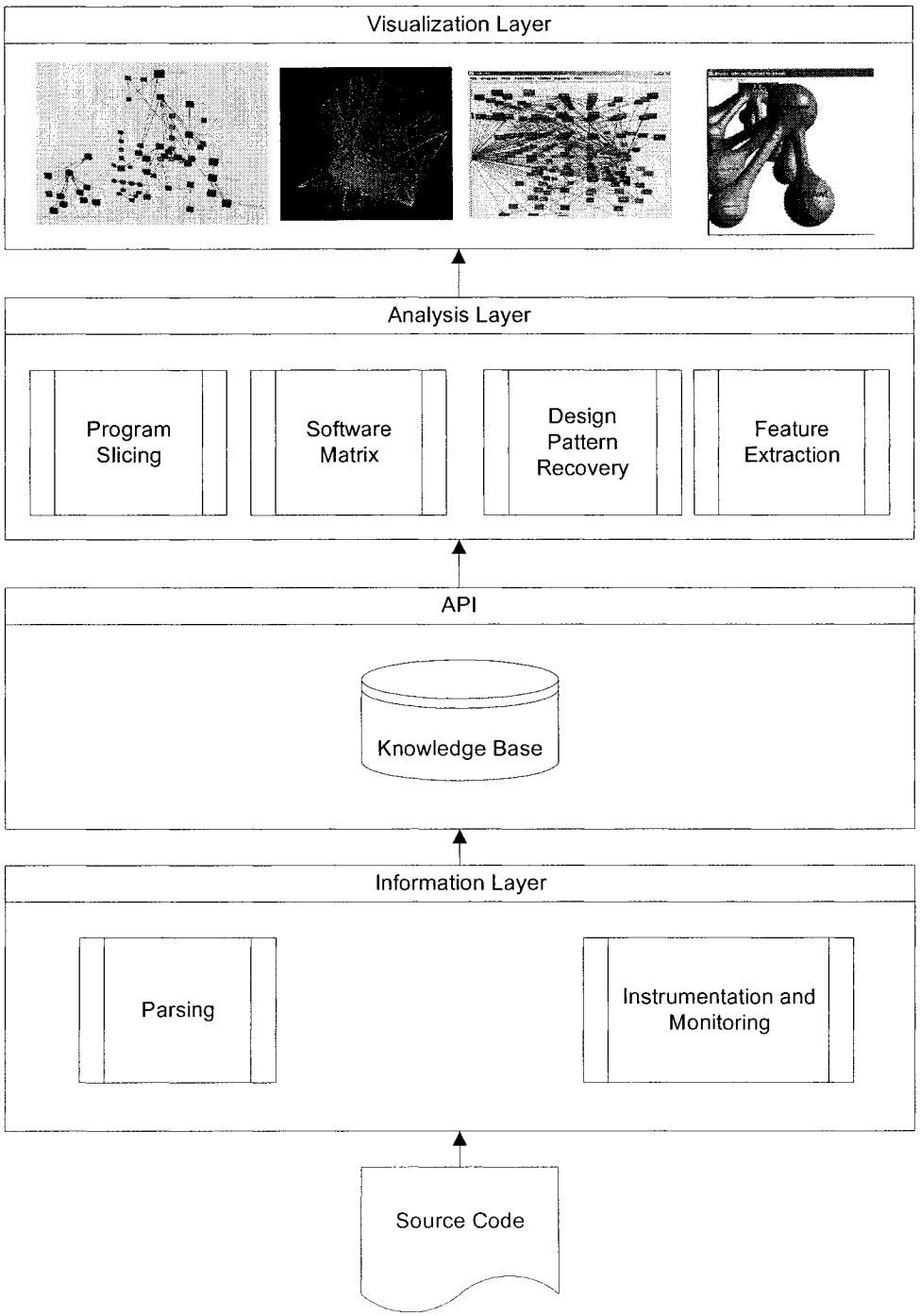


**Figure 22: CONCEPT architecture, from source code to 3D visualization [RILL02b].**

Figure 23 presents the steps required for the extraction of design pattern from source code. In this research we only consider design patterns based on an existing design pattern recovery tool [ZHA03] that was previously developed as part of the CONCEPT project. **3D-UMLVis** is provided with design patterns present in the code along with the classes that participates in them. **3D-UMLVis** groups the classes together in virtual 3D space, then presents the information to the user via a 3D rendering environment.

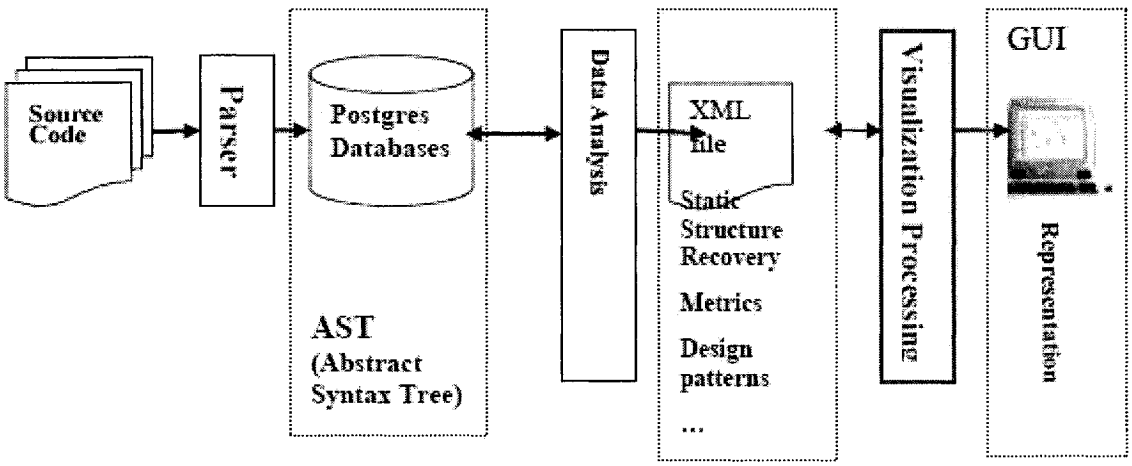


**Figure 23: Steps in extracting design patterns from source code.**

This research contributes mainly in the 'Visualization Processing" portion of the overall system. In this part the extracted design patterns are imported from an XML file and then laid out with respect to the representations suggested by the GoF. After the layout for the design patterns is completed the "Representation" takes charge of displaying, viewing and navigating.

## 3.5 Approach

### 3.5.1 Source code parsing

Source code parsing, design pattern detection and layout are works from previous research [HEUZ03, RILL02b, SHAU98, NOBL03]. Although the research is still in progress, the **3D-UMLVis** is provided with a data file in XML format in which a layout has already been made and the design patterns detected. **3D-UMLVis** depicts design patterns that exist within the provided XML data file and then renders the classes and the existing layout in the 3D space.

### 3.5.2 Design patterns layout

Design patterns are increasingly accepted as a major concept within the software engineering domain.

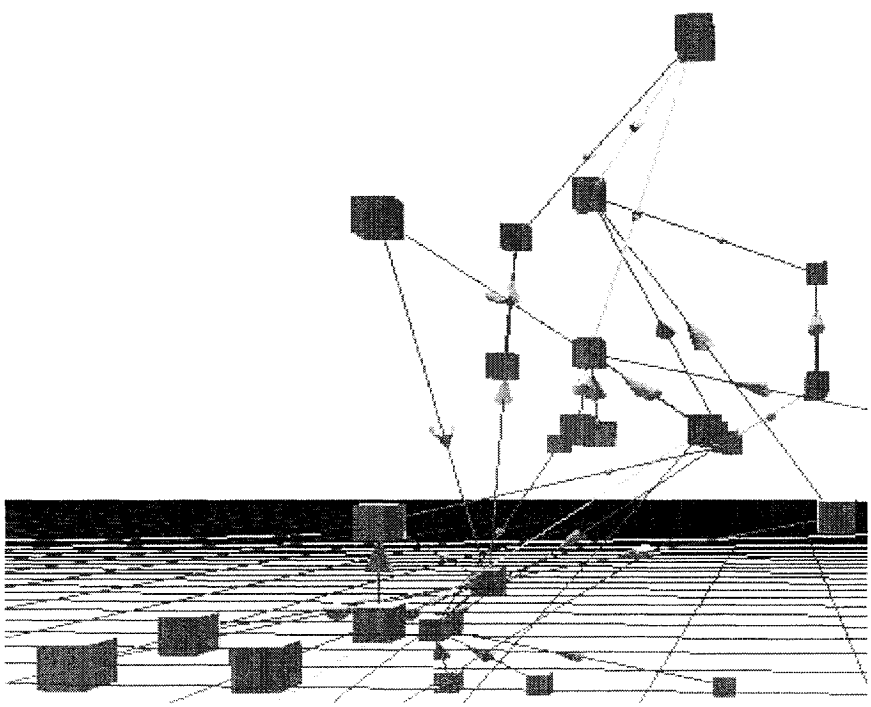

**Figure 24: A view where classes of a design pattern are scattered all around the view.**

In the context of this research **3D-UMLVis** applies an existing source code analysis tool [HEUZ03] [SMIT03] to detect design patterns that were implemented during the forward engineering process.

Design patterns are typically documented in 2D UML and are fairly easy to recognize. However, most of the existing tools and even the UML standard do not support the visualization of 3D UML design patterns. Under a 3D environment, when elements are not grouped coherently, they quickly become meaningless. Figure 24 presents design patterns that are ungrouped, classes are scattered and it is difficult to locate any design pattern presence.

However, like a star constellation, design patterns become recognizable when they are coherently and consistently grouped, even when mixed amongst other classes. The goal is therefore to recreate a layout of recovered classes and patterns that matches the design pattern layout proposed by the GoF. The resulting view will be more meaningful and allows visual pattern matching and recognition. Therefore, the interpretation of classes and their respective roles in the overall design becomes more intuitive since they now match a layout generally accepted by the software engineering community.

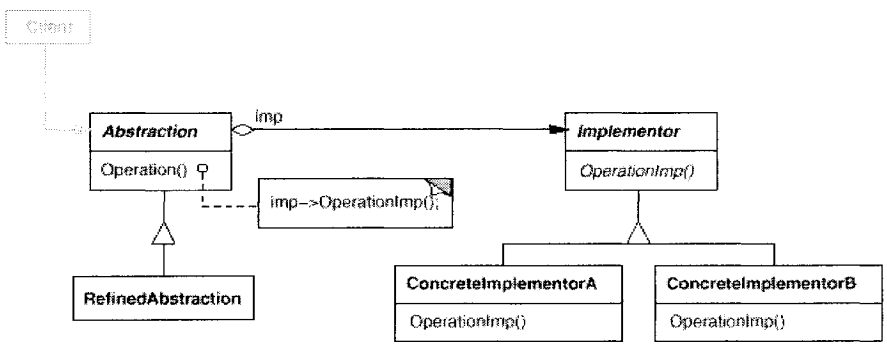


**Figure 25: The well-known and easily recognisable bridge pattern.**

Familiar shapes and formations are more recognizable. However, 2D representations have drawbacks when the design pattern has more than one class with the same role, which is likely the case in any software. Take for example bridge pattern in figure 25 representing a bridge pattern. If one were to add four more classes playing the role of "concrete implementor" to the pattern, then the traditional layout must be distorted. Attaching those classes will add complexity to the layout. Figure 26 shows that the ideal pattern layout must be rearranged to accommodate extra members in a 2D viewing environment.

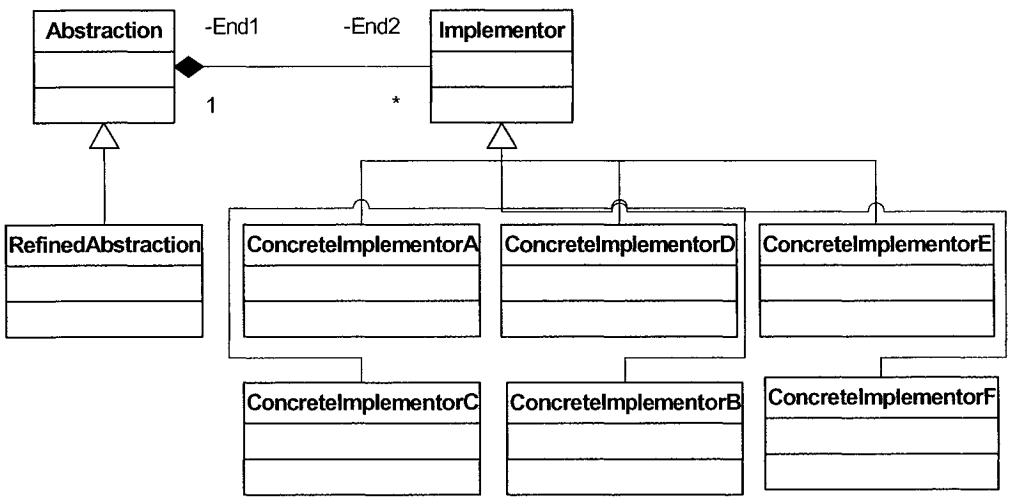


**Figure 26: A design pattern with extra members.**

Traditional 2D representation would have to position classes with the same role one "beside" each other (Figure 27) which makes the view cumbersome and nowhere resembles the layout proposed by the GoF.

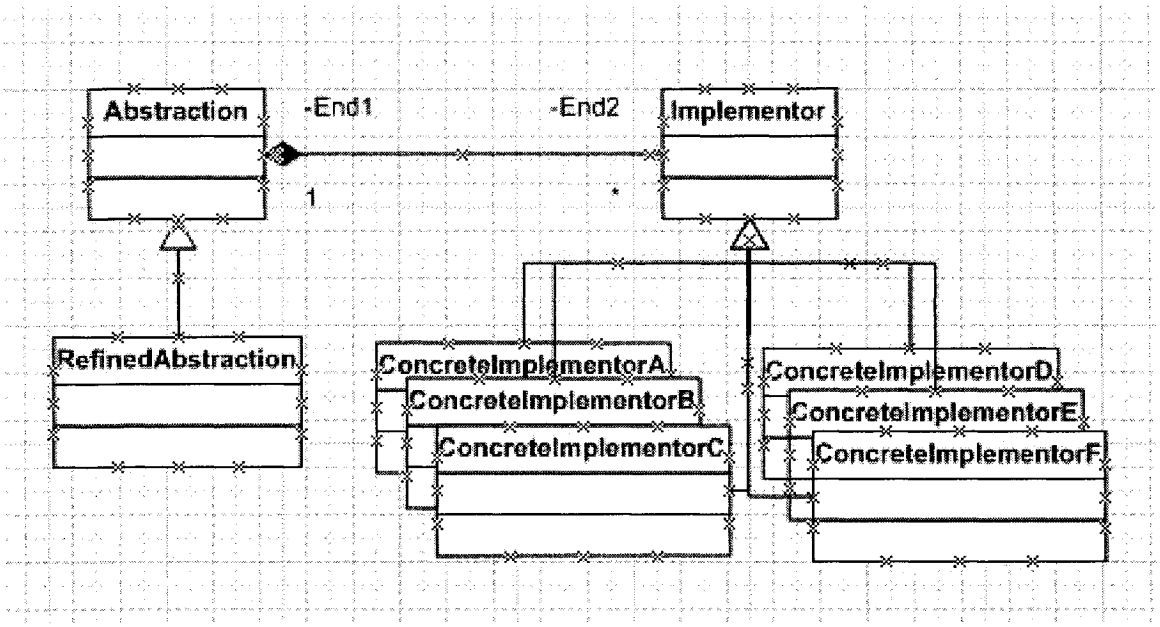


**Figure 27: The bridge pattern with classes of the same role "stack" behind each other.**

On the other hand, a 3D representation can "stack" those same classes one "behind" each other and the result is a more compact layout that requires less screen space, and still respects the layout template proposed by the GoF. The user can clearly see that these classes are part of the bridge pattern without having to read a single line of code as shown in Figure 27.

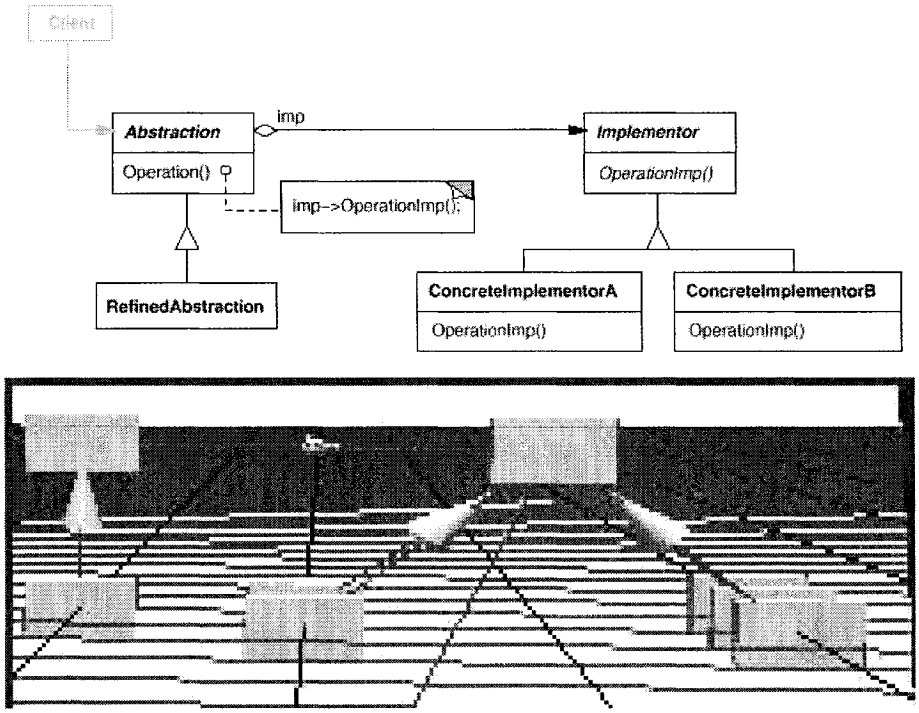


**Figure 28: Classes reorganised with respect to their design pattern.**

As an example, Figure 28 shows how **3D-UMLViz** can rearrange and groups classes together, allowing for a much cleaner view of the design pattern. Classes sharing the same role in the design pattern are "stacked" behind each other to preserve the view proposed by the GoF.

### 3.5.3 Dealing with Design Pattern Crosscutting

It is common that a class participates in more than one design pattern (often also refereed to as "crosscutting"). In a simple 2D layout view, classes that participate in different design patterns would be connected by crossing edges among the different patterns and/or the layout that would reflect the GoF layout is distorted, by moving classes from one pattern to another one. As a result of this crosscutting, the same design

pattern is now more difficult to detect in the diagram even though the classes participating in that pattern are grouped together as shown in the Figure 29. In such a situation The layout is confusing and one must also choose a design pattern with which to group the algorithm.

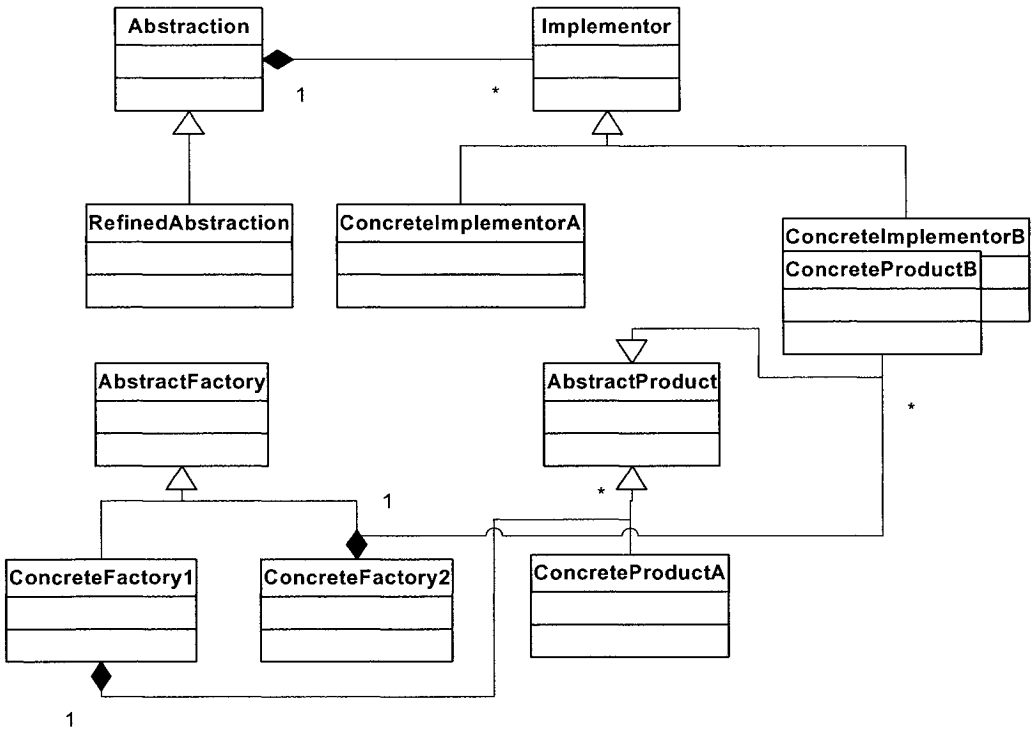


**Figure 29: Crosscutting between "Bridge" and "Abstract Factory" pattern.**

A two step approach is taken in developing **3D-UMLViz**. In the first step, the layout algorithm identifies a location in 3D space where to position and group the classes that belong to a design pattern. The GoF design pattern templates are used to position and group the classes participate in a design pattern according to their role. In the second step, two-view options are used to address the issue of crosscutting between design patterns. The first view mode is a *compact* mode where classes that participate in the same design patterns are grouped together and classes that participate in more than one

patterns will be assigned to one of the patterns. In the *compact* mode, **3D-UMLViz** arbitrarily assigns the class to one design pattern configuration whereas the other design pattern will be missing a member. This view generates some patterns with missing members but the view will not introduce any additional class artefacts. The drawback is that the design pattern in the *compact* mode might not always be recognizable due to a missing object in the layout.

The second view mode is the *relaxed* mode. It is similar to the *compact* mode, but classes that participate in multiple design patterns will be duplicated in all the design patterns. This mode will add to the complexity of the diagram, since the same class can appear in multiple locations. On the other hand, it allows the tool to lay out all the design patterns in a fashion suggested by the GoF template.

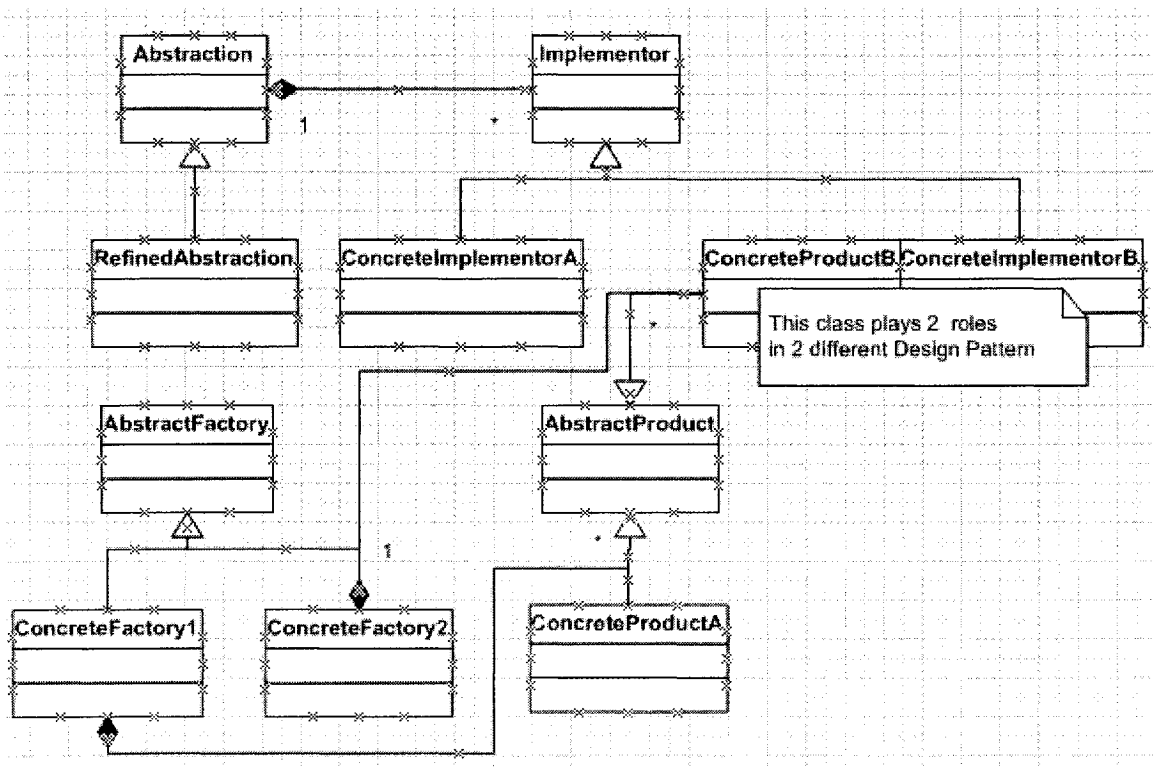


**Figure 30: Crosscutting in the *compact* mode.**

Figure 30 shows an example of a *compact* mode, where a class is arbitrarily assigned to one pattern and the second pattern is missing a member.

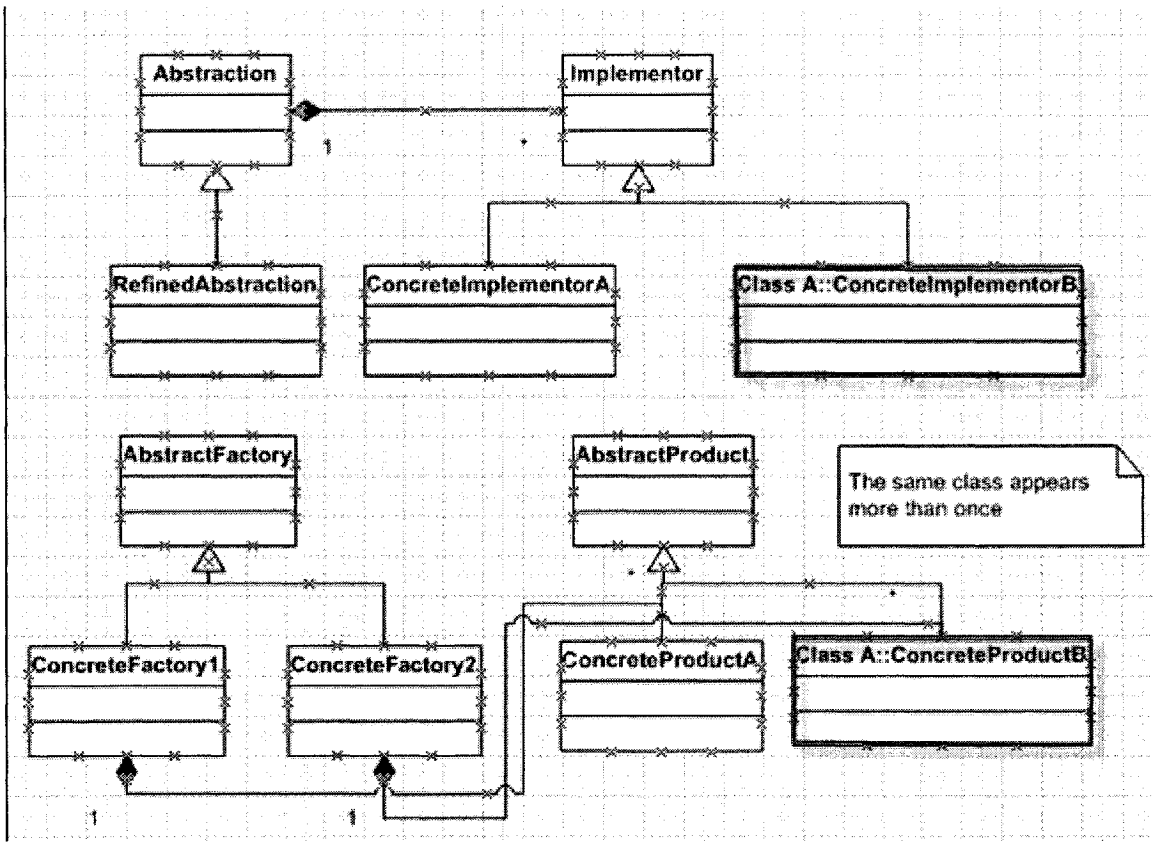


**Figure 31: Crosscutting in the *relaxed* mode.**

Figure 31 shows the *relaxed* mode, where a "Class A" may appear in more than one location if it participates in multiple design patterns.

Switching between these two modes can cause some disorientation for the user, due to the change of the context in which the diagram is viewed. An immediate switch will cause some classes to disappear and fragments to appear, and *vice versa*. This issue is addressed in **3D-UMLVis** through animation that allows closing this context gap between the two views. The animation provides the user with a visual link between the

two modes. An animated transition creates a gradual and continuous transition between the two views, by animating the repositioning of the classes from one view to another. Instead of seeing a sudden change, the user can now see a gradual transformation from one diagram to the other. The animation shows the class that participates in multiple patterns break down into multiple fragments and each piece transit toward the positions where is should reside for each pattern. This animation works in both directions, from the *compact* to the *relaxed* (class break down into fragments) mode and *visa versa* (fragments collapses back together).

### 3.5.4 Navigation in a 3D scene

Two-dimensional navigation is typically restricted to scrolling up, down, left and right. In 3D navigation additional challenges and benefits can be identified due to the added depth navigation. Benefits of 3D navigation include the ability to implement a navigation that is more intuitive to the user such as walk through, fly through and rotation.

However, it must be noted that a poorly designed three-dimensional navigation system can cause a loss of perspective and disorientation. Navigating in 3D space sometimes causes total disorientation and the user must then "reset" the view and start over again. Ideally, the user would not want to do this since resetting the view might also mean restarting the comprehension process. A more intuitive navigation system with reference markers will help the user to locate and self-orient in the scene. Such a navigation system will resemble a helicopter flying over the earth surface rather than a

plane through space. In comparison, a plane can roll whereas a helicopter cannot. Rolling cause disorientation since the user tends to lose the notion of up and down. **3D-UMLVis** proposes an overview map and an indicator of the user's current location to avoid being lost in the virtual space.

Within **3D-UMLVis**, a solution proposes a global view of the entire diagram that shows the current location of the viewpoint and the direction that the user is currently viewing. This approach is similar to a global positioning system. The user can always pinpoint his location within the 3D space.

Furthermore, navigation can be confusing in 3D space if the user does not have a notion of what is up and what is down. In real world navigation, the notion of up and down is clear (defined by gravity and the surface), but in 3D virtual space, that notion no longer exists. **3D-UMLVis** compensates the lack of gravity and surface, by adding a virtual ground in the diagram to create a sense of up and down.

The last navigation challenges are the input devices. Navigation devices should support the notion of a 3D space in the form of X, Y and Z coordinates. Typical mouse or key pointers do not allow this 3D navigation. Other devices, more specific for 3D, input such as data gloves, gaming input devices (joysticks, wheels, etc) should be considered for further improvement. These 3D specific input devices were not considered in this research.

**3D-UMLVis** implements a combination of mouse and keyboard inputs based on the navigation concepts typically supported by computer games. It requires the user to "move" (translate) the viewpoint with the keyboard and "look" (rotate) with the mouse.

The effects are similar to a human walking and looking around by turning his head. This metaphor is closer to the real world and thus easier to grasp.

# Chapter 4 Implementation

## 4.1 The CONCEPT environment

As mentioned earlier, the CONCEPT project [RILL02b] supports both source code analysis and design pattern recovery techniques. The project is part of an ongoing effort to facilitate program comprehension and improve software maintenance. The CONCEPT environment is currently being implemented as a plug-in to the ECLIPSE platform [ECLIPSE]. All its tools and functionalities can directly be triggered or launched from the ECLIPSE environment. Following the same approach, **3D-UMLVis** is integrated with the CONCEPT and ECLIPSE environment. Therefore, after the parsing and extraction process is completed, **3D-UMLVis** can be applied to view both, the class models extracted from the source code and the recovered design patterns.

OpenGL [OGL] was chosen as the graphic library since it is mostly independent of the platform and the operating system. All functions for windowing task as well as functions for user input are excluded, making it easy to combine OpenGL with other platform-dependant programming libraries that handle the windowing and user input. Furthermore, OpenGL is a streamlined, high performance graphic rendering library. The design is close to the graphic hardware to render geometric objects such as points, lines and polygons at tremendous speed. Consequently, with efficient code, the tool can render hundred of thousands of polygons without compromising the frame rate. This is especially important when the size of the diagram increases and the number of objects can drastically slow down the rendering process.

OpenGL library also includes functions for some important 3D properties, like scene camera, lights, textures, face culling etc. The OpenGL Utility Library (GLU) contains all routines that use lower-level OpenGL commands to perform matrices for specific viewing orientations and projections, polygon tessellation and surface rendering. The extension to the X-Windows system (GLX) provides means of creating OpenGL context and associating it with a window that can be drawn on machines using the X-Window system. The Utility Toolkit (GLUT) functionalities support multiple window rendering, call back driven event processing, "idle" routines, timers and various solid and wire-frame basic objects rendering. With all the support that OpenGL provides and its portability, it is the ideal rendering system for the prototype GUI.

## 4.2 XML data parsing.

The CONCEPT project provides an implementation for static source code analysis [ZHAN04] that detects and recovers design patterns from source code. It also has a grouping algorithm and a layout algorithm to generate the 3D layout for the "Virtual city" [SHI04]. The data input parsing is rather simple since all artefacts and their respective coordinates for the 3D space are provided. There exist many free XML parsers that can provide a simple interface to access the elements of the XML tree data. A sample XML data input is presented in Figure 32.

```
<entities>
    <!--Abstract factoy pattern-->
    <entity id="0" name="Abstract_Factory::CFactory" material="METAL" visible="true" >
        <size factor="1" controlRadius="1"/>
        <coordinator x="-2" y="5" z="0"/>
    </entity>
    <entity id="1" name="Concrete_Factory::CGraphicObj" material="PEWTER" visible="true">
        <size factor="1" controlRadius="1"/>
        <coordinator x="-3" y="3" z="0"/>
    </entity>
    <entity id="2" name="Concrete_Factory::CRealation" material="PEWTER" visible="true">
        <size factor="1" controlRadius="1"/>
        <coordinator x="-1" y="3" z="0"/>
    </entity>
    <entity id="3" name="Abstract_Product::C3dObj" material="PEWTER" visible="true">
        <size factor="1" controlRadius="1"/>
        <coordinator x="1" y="5" z="-1"/>
    </entity>
    <entity id="4" name="Abstract_Product::CArrow" material="PEWTER" visible="true">
        <size factor="1" controlRadius="1"/>
        <coordinator x="1" y="3" z="-1"/>

<!--The observer pattern-->
<entity id="55" name="Subject::CData"> <coordinator x="-13" y="2" z="3"/>
<entity id="56" name="ConcreteSubject::CTxtData"> <coordinator x="-13" y="1" z="3"/>
<entity id="57" name="ConcreteSubject::CXMLData"> <coordinator x="-13" y="1" z="2"/>
<entity id="58" name="Observer::CSceneView"> <coordinator x="-11" y="2" z="3"/>
<entity id="59" name="ConcreteObserver::CMainView"> <coordinator x="-11" y="1" z="3"/>
<entity id="60" name="ConcreteObserver::CCompactView"> <coordinator x="-11" y="1" z="2"/>
<entity id="61" name="ConcreteObserver::CRelaxedView"> <coordinator x="-11" y="1" z="1"/>
<entity id="62" name="ConcreteObserver::CSummaryView"> <coordinator x="-11" y="1" z="0"/>
```

**Figure 32: Sample XML data input.**

The data also provides a list of classes that participate in a design pattern. Although those classes are already grouped close to each other in 3D coordinates, they are not positioned in a way that would match with the GoF layout. The **3D-UMLVis** uses the GoF layout as a template to reorganize the classes that corresponds to the basic UML layout [BOOC99]. Using a simple grid algorithm, **3D-UMLVis** relocates the classes to their new positions resulting in a layout specific for the particular pattern. There is a separate sub-layout algorithm for each of the specific GoF patterns.

## 4.3 Visualizing design patterns.

When the design patterns are presented in a traditional 2D UML visualization suffers some drawbacks as previously discussed in section 3.3. Within the **3D-UMLVis** classes that share the same role are stacked in a design in view of preserving the original layout proposed by the GoF.

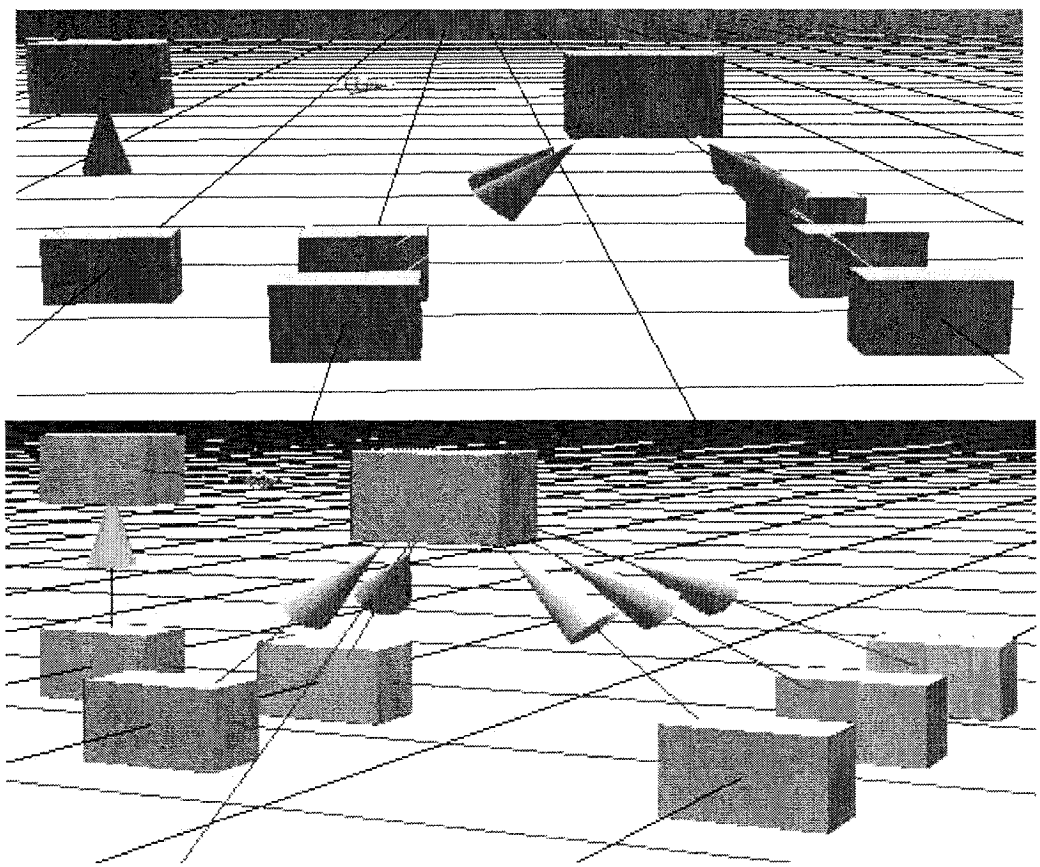

**Figure 33: Classes "stacking".**

Figure 33 illustrates such a class stacking, while preserving the "Bridge" pattern layout. Not only is the GoF layout preserved, **3D-UMLVis** can also accommodate multiple classes serving the same role in the design patterns. Figure 33 also illustrates a view from the front, as well as from a different angle, a feature only available in 3D.

This grouping layout allows the user a visual pattern matching of the familiar design pattern shape/layout, even in a busier class diagrams.

## 4.4 Dealing with crosscutting between design patterns

As described in section 3.5.3, the major problem in grouping classes with their respective design patterns is that some classes might participate in more than one design pattern, known as crosscutting. One class can play the role of a "concrete product" in an "Abstract factory" pattern and the role of a "concrete implementor" in a "Bridge" pattern. One must decide where to position the class. Should it be grouped with the abstract factory pattern or should it be grouped with the bridge pattern?

The situation will cause one of the pattern layouts to be incomplete. A class can only be physically be at one location at a time. 3D provides the advantage of a depth layer, animation and the ability to follow gradual animated changes with multiple camera views. Countless viewing angles are available in 3D without having to reorganize the diagram. While it is difficult to find one view angle that suits all comprehension tasks, 3D allows the observer to choose the view that best fit the current requirement. The 3D crosscutting problem is circumvented by providing two alternative views.

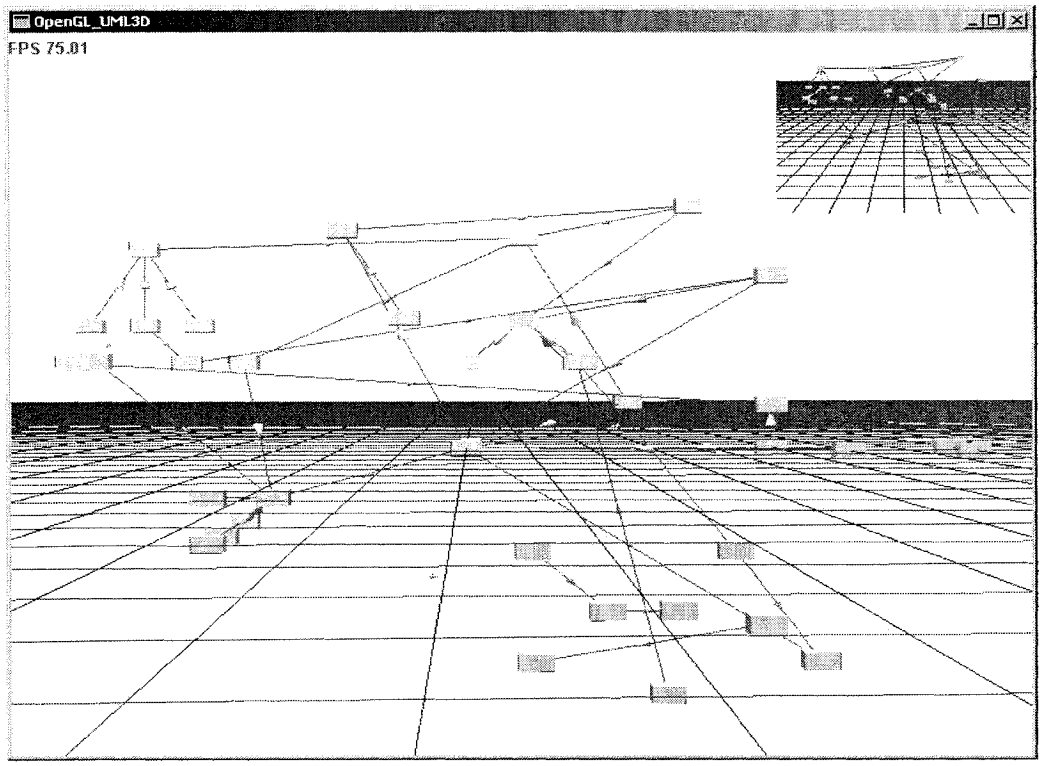


**Figure 34: 3D-UMLVis *compact* view mode**

The *compact* view mode groups classes with their respective design patterns explained previously in section 3.5.3. However, in cases of crosscutting a class cannot be grouped within all of its respective patterns. There is only one instance of each class in the view, and the pattern might not be clearly identifiable. The *compact* mode is typically what **3D-UMLVis** can do best in grouping design patterns. Classes of the same design patterns are physically grouped together, see Figure 34. However, some patterns will be missing some members from its configuration. This causes some difficulties with recognizing the design pattern on a crowded environment since its shape is not familiar.

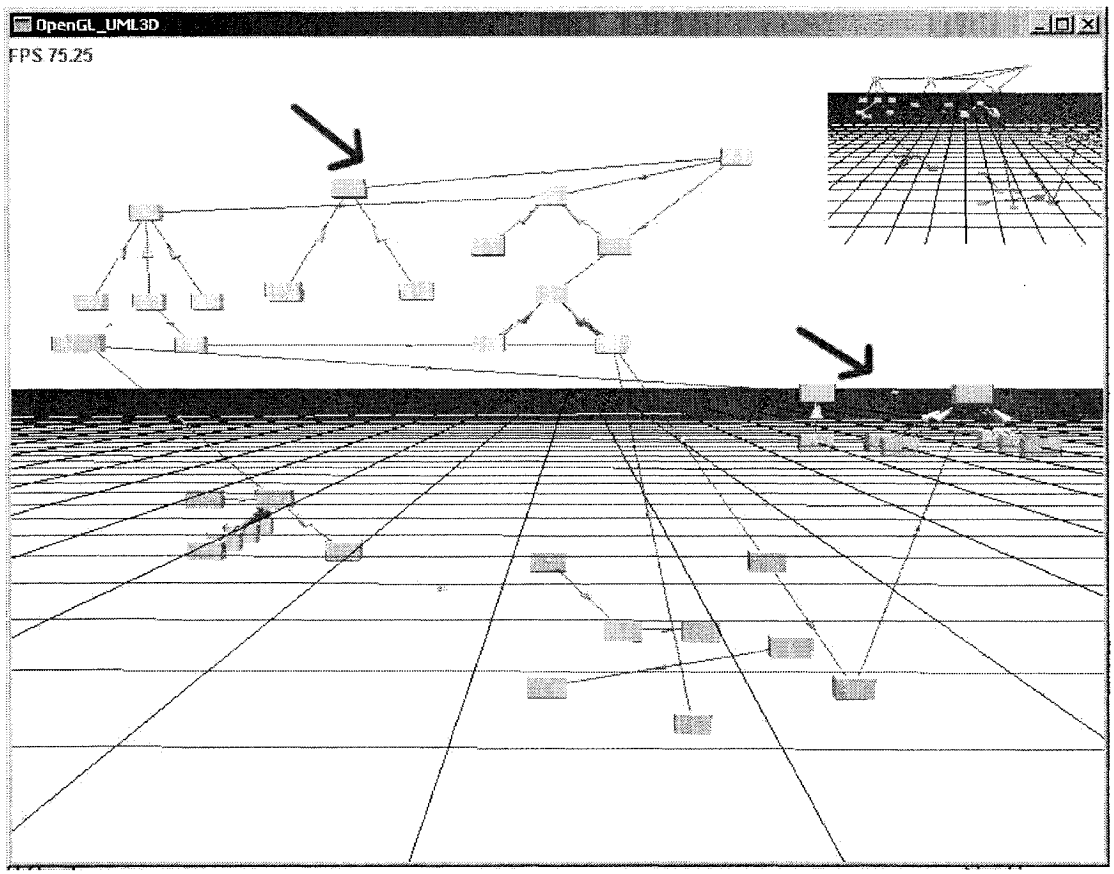


**Figure 35: 3D-UMLVis *Relaxed* view mode**

Figure 35 shows the same diagram in the *relaxed* view mode. In this mode, the design patterns are positioned according to the layout presented in [BOOC99]. However, the price for following the GoF template is that one has included extra artefacts in the *relaxed* view. Classes that participate in more than one pattern will appear several times (depending on their number of occurrences in different patterns). Each copy will be placed in the layout location predetermined by their respective role in the pattern. A drawback of this approach is the number of software artefacts and complexity of the diagram increases. This is one of the main reasons why both of *relaxed* view mode and *compact* view mode are provided in the development of **3D-UMLVis.**

## 4.5 Smooth transition between *Compact* and *Relaxed* mode

In general, switching between the different view modes is a problem in software visualization. The user might easily get disoriented and might loose the context in which a diagram was viewed. **3D-UMLVis** introduces an animated transition where the classes that participate in several design patterns will be "split out" in several fragments and "migrate" toward the position they occupy in the pattern. A wire-frame copy denoting the migration end point indicates the destination of each "class split". The animation creates a smooth transition between two end points and allows the user to visually follow the transition, see Figure 36.

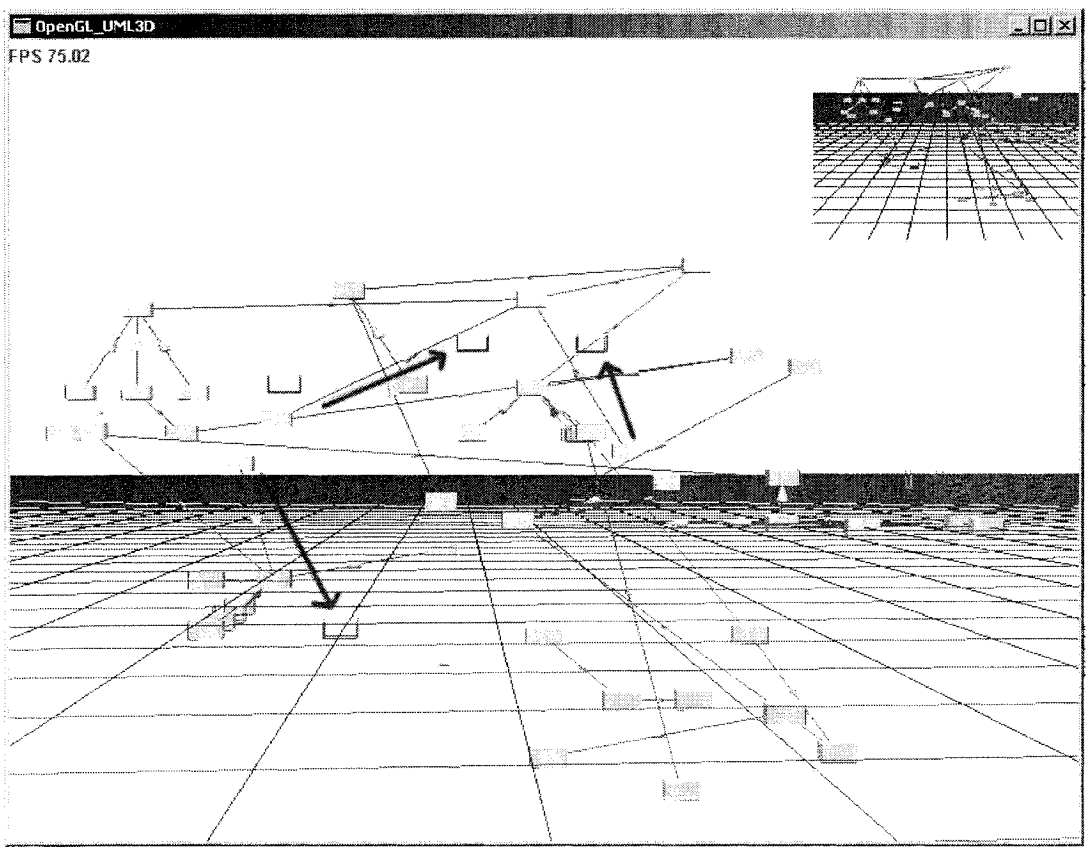


**Figure 36: Animated crosscutting, classes migrates to their assigned position.**

The user can follow the diagram transformation from one view mode to the other and therefore reducing the disorientation factor. The camera can also focus on the migration of a single class. The camera will zoom in to the class of interest and follow its migration path from one destination to another. The user can also freely alternate between the "*Compact* view mode" and the "*Relaxed* view mode". A similar animation is supported for any diagram zoom in/out, allowing the user to follow the view without loosing focus and orientation.

## 4.6 Addressing the Navigation Challenge

There are differences between the visual structure of 2D and 3D interfaces. In most situations, 3D interfaces do not offer a unique, comprehensive view. The user needs to move through the scene to perform a given tasks. 3D interfaces are generally based on exploration rather than on synthesis, that is the reason why orientation and navigation are among the key issues for a successful 3D interface [NEIL98].

Orientation is the first requirement for navigation. A proper orientation strategy will help the user navigate the virtual 3D environment. There are a several techniques for supporting orientation within a 3D environment:

- Signs to communicate directions, in terms of the 3D world, **3D-UMLVis** can use x/y/z axes to aid orientation.
- Tools that identify the current position with respect to natural or artificial key points. In this 3D world, **3D-UMLVis** applies a global positioning system of the scene and an identification to locate the user's current position.

- A global plan of the scene that allows the user to locate himself within the big picture, a map to indicate where he is.

**3D-UMLVis** has added to the 3D scene a ground grid to help with orientation, to provide guidance whether one is viewing the scene from a top angle (above the ground level) or from below. Within the multiple window system, **3D-UMLVis** simultaneously provides a general global system view always at the same level of abstraction. Therefore the global view can be used as an orientation guide to recover the previous location if one navigates within the detailed view. Once successfully positioned and oriented, the user can then start navigating through the 3D scene. However, this task is often more complicated than navigating through a real world environment. Within the software environment the user is limited to the available input devices (mouse, keyboard, game-pad, trackballs etc.) each of which alone is insufficient. The key is to combine these inputs, to provide a multimodal approach and to have devices that complement one another [MALE01]. Light is also used to help orienting the user. In the real world, people use sunlight and shadow to orient themselves when other method fails. Similarly in the virtual word, **3D-UMLVis** uses lighting techniques to help users during the orientation process.

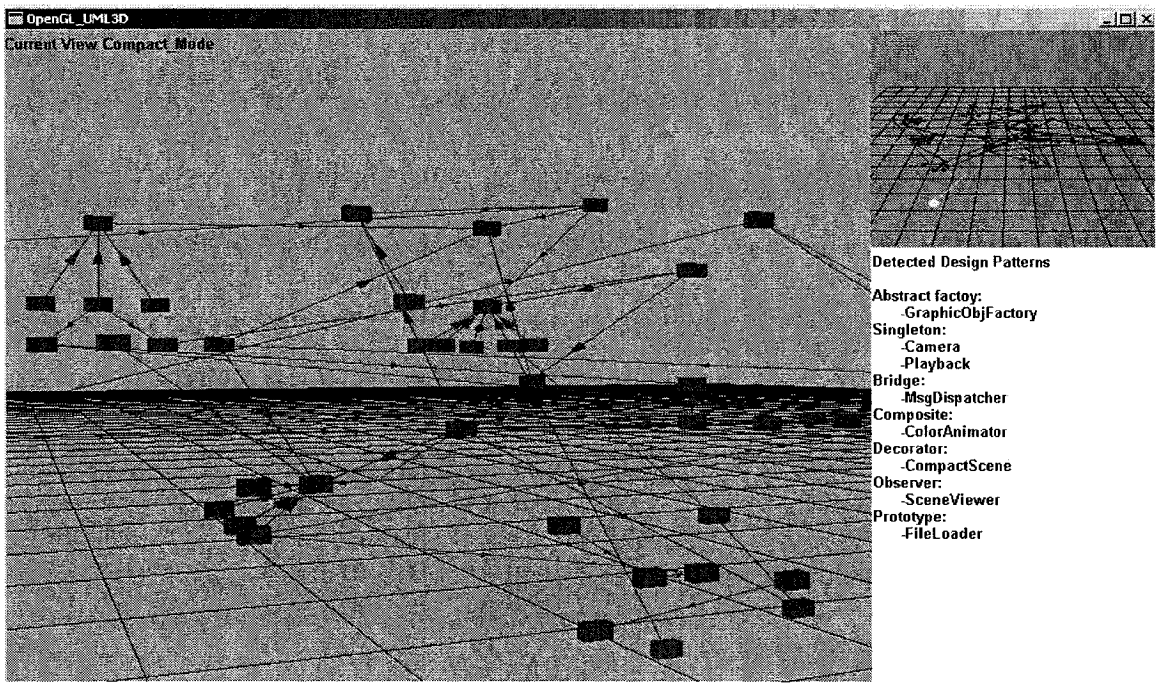


**Figure 37: Navigation system in 3D-UMLVis**

Figure 37 shows the overview window on the top right, in which the current user position and viewing angle is indicated by a dot. This marker shows the current location and direction of the viewing point so the user can determine where he is in the 3D world. Moving the mouse will rotate the viewing direction, simulating the user looking around like a person turning his head to look around his environment. Movement is accomplished through the directional arrows on the keyboard where up/down is used as moving forward/backward and left/right is used as "side-stepping". This navigation technique simulates a person walking around the 3D word to explore its components. This technique endeavours to submerge the user in his virtual environment.

## 4.7 Integration within ECLIPSE

Since **3D-UMLVis** requires pre-processed information (to generate the XML input), it is essential that a user can perform operations such as selecting source code to be parsed, generate the XML input data and then visualise the result in **3D-UMLVis** all within one single tool. As mentioned in section 4.1, **3D-UMLVis** would be an extra plug-in into the ECLIPSE environment and thus completing the CONCEPT process of parsing, extracting and then viewing the design patterns. Figure 38 shows how **3D-UMLVis** plug-in integrates with the ECLIPSE interface.

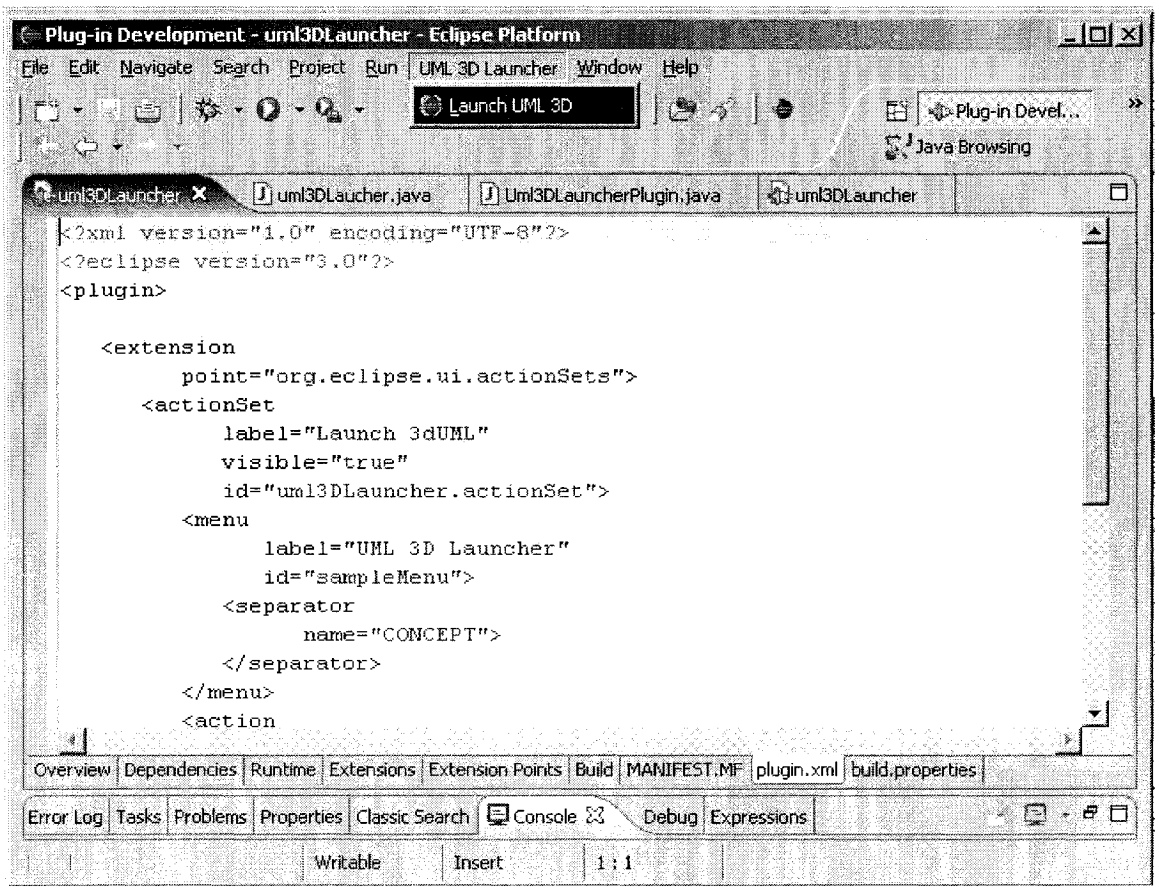


**Figure 38: Integration of 3D-UMLVis as an Eclipse plug-in.**

## 4.8 Related work and discussion

It has been shown that displaying UML diagrams in 3D is useful and feasible [DWYE01, MCIN04]. As mentioned in the previous sections, taking advantage of the third dimension allows for extra flexibility to increase the expressiveness and comprehension of UML diagrams. With the rapid advances in computer graphics and the increased performance of video cards, one can now display several millions of polygons per second allowing for the display of large systems without compromising the rendering frame rate [NEIL98].

The core of the system is the knowledge base that stores both static and dynamic information extracted from source code, execution traces and available documentation. The knowledge base serves as input to the various analysis techniques supported by the environment. The results can then be visualized through different visualization techniques integrated within the viewing environment.

UML, design patterns and 3D visualisation are topics of sustained research interest [LEWE01, FOUR04, DONG03] and researchers recently began combining the advantages of 3D visualization and UML representation. As demonstrated in [THAD03], 3D animation and UML representation can greatly help the comprehension of object-oriented programs. The animation helps understand the dynamic sequence of program execution, which means that programmers have an intuitive model of the software actions. [SHI04] has also demonstrated that one could help the comprehension process by representing design patterns in a 3D environment. Under the metaphor of a virtual city, users can grasp the interactions between classes participating in a design patterns.

**3D-UMLVis** combines the concept of 3D visualisation, 3D animation and UML design patterns representation to address a new aspect of design patterns. It addresses the cross-cutting between design patterns, the event where a class participates in multiple design pattern concept. **3D-UMLVis** fully takes advantage of the third dimension from the 3D environment to "stack" classes that plays the same role within a design pattern thus keeping the same layout presented by the GoF. This approach is not limited to reverse engineering design patterns; it can also be applied during forward engineering, while creating class models. However, 3D-UMLVis still cannot address information overload, 3D texts and view scalability which are the major hurdles of Software Visualization.

Some of the remaining challenges and problems are:

1. Grouping and layout depend on the availability of source code analysis techniques and the quality of the design pattern recovery. Presently the CONCEPT project is limited to the recovery of eight static GoF patterns.
2. Scalability continues to be an important issue. The existing screen space is still a limiting factor for really large systems. Further filtering and use of other abstraction levels will be needed.
3. Current approaches are limited by the UML metaphor itself. There is a lack of expressiveness to represent certain semantic grouping/information such as design patterns, features, etc.
4. Domain knowledge plays an essential role during the reverse engineering process. It is an inherently complex problem of reverse engineering, to recover and therefore also to represent domain knowledge that had an influence during the

forward engineering. The expertise of the programmer using the tool and his/her ability to identify what grouping and/or analysis techniques might best match the original design determines the usefulness of the presented approach.

5. Layout algorithms have to be adapted individually to the different design patterns. There is no general layout algorithm for all design patterns.

# Chapter 5 Conclusions and Future Work

Numerous techniques and metaphors to visualise software system are available today. Several of these techniques go beyond the traditional approach of 2D representation by utilizing and taking advantage of the 3D space. However, most of the existing tools and even the UML standard do not support the visualization of UML patterns.

This thesis reviews major software visualization techniques and tools. A discussion on the use of a 3D representation for UML and how this representation in combination with source code analysis can benefit the comprehension process is presented. More specifically issues related to crosscutting, navigation and the use of animation to visualize recovered design patterns from the source code are addressed. It should be noted that this thesis does not claim that there exists one solution to all problems. The effectiveness of the presented approach of combining 3D visualization with semantic source code analysis depends on several factors that limit its applicability. Furthermore, techniques for visualizing design patterns in the 3D space are presented, as well as challenges with respect to cross-cutting and navigation were discussed and addressed. In addition, new viewing techniques are proposed to facilitate the navigation and filtering of the information in this 3D world. A tool, entitled **3D-UMLVis**, was implemented as a part of this research to illustrate the applicability of the presented techniques.

Future work should address issues related to extending the analysis techniques for detecting design patterns. Another major challenge which should be addressed as part of future work should be identifying new metaphors that are more suitable to represent concepts like design patterns and features. The challenges will not be to identify new metaphors only but also to ensure that these new metaphors become an integrated part of the modeling process (forward and reverse engineering). Regrettably, this thesis could not address fully the issue of scalability. Design patterns formations can easily be recognised within a large diagram but as more artefacts are added to the view, the configuration is less visible. It would require additional filtering and grouping strategies to tackle this issue of large systems. The CONCEPT project explored the use of slicing to produce a smaller visualisation. However, in several occasions, the resulting slice still includes too much information to be usable, even for visualization in 3D.

## References:


[1] [ANTO98] G. Antoniol, R. Fiutem and L. Cristoforetti "Design Pattern Recovery in Object-Oriented Software", In Proc. of the 6th IWPC'98, pp 153-160, Ischia, Italy, June, 1998.

[2] [BALL96] Ball T., Eick Stephen G., "Software Visualization in the Large". *IEEE Computer* 29(4): 33-43 (1996).

[3] [BART94] L. Bartram, R. Ovans, J. Dill, M. Dyck, A. Ho, and W.S.Havens, "Contextual Assistance in User Interfaces to Complex, Time Critical Systems: The Intelligent Zoom." Proc. Graphics Interfaces'94, 216-224. 1994.

[4] [BASS02] Bassil S., Keller R.K. "Software Visualization Tools: Survey and Aanlysis", Proc. IEEE 9th Intenat.Workshop on program Comprehension (IWPC'01), 2002, pp 7-17.

[5] [BECK94] K. Beck, R. Johnson, "Patterns generate architectures", In Proc. of the 13th European Conference on OO Programming, Lecture Notes in CS Nr. 821. 1994.

[6] [BEDE94] B. Bederson, and J. Hollan, "Pad++: A Zooming Graphical Interface for Exploring Alternative Interface Physics", ACM UIST Proceedings, ACM Press,17-26 1994.

[7] [BELA71] L.A. Belady, M.M. Lehman. "Programming system dynamics or the Meta-dynamics of system in maintenance and growth." Technical report RC 3546, International Journal of Man-Machine Corporation, 1971.

[8] [BOEH81] Barry W. Boehm. Software Engineering Economics. Prentice Hall, 1981

[9] [BOOC99] G. Booch, J. Rumbaugh and I. Jacobson, "The Unified Modeling Language Reference Manual", Addison Wesley, 1999.

[10] [BROO77] Brooks, R., "Towards a Theory of the Cognitive Processes in Computer Programming", *Int. J. Man-Machine Studies*, Vol.9, 1977, pp.737-751

[11] [BROO83] Brooks, R., Towards a Theory of the Comprehension of Computer Programs, *Int. J. of Man-Machine Studies (18)*, pp. 543-554, 1983.

[12] [BROO87] F. Brooks Jr. No silver bullet. Computer, 1987.

[13] [DEER92] M. Deering, "High resolution virtual reality. Proceedings of SIGGRAPH'92. In Computer Graphics", 26, 2 195-202. 1992.

*[14]* [DONG03] Jing Dong and Sheng Yang, "Visualizing Design Patterns With A UML Profile", In the Proc. of the IEEE Symposium on Visual/Multimedia Languages (VL), Auckland, New Zealand, October 2003, pp123-125

[15] [DWYE01] Dwyer T., "Three Dimensional UML Using Force Directed Layout", *Proc. Australian Symp on Inf. Visualization*, 01.

[16] [ECLIPSE] www.eclipse.org

[17] [EISE] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon, *Locating Features in Source Code*, IEEE Trans. on Software Engineering, Vol. 29, No. 3, pages 195-209.

[18] [FAVR01] Favre J.M., "G$^{SEE}$: a Generic Software Exploration Environment", *9th. Workshop on Program Comprehension (IWPC'2001),* Toronto, May 2001, pp. 233-244.

[19] [FERE01] R. Ferenc, J. Gustafsson, L. Muller, J. Paakki, "Recognizing Design Patterns in C++ programs with the integration of Columbus and Maisa", In Proc. 7th Symposium on Programming Languages and Software Tools (SPLST 2001), Hungary, June 15-16, 2001. pp 58-70

[20] [FJEL83] R.K. Fjeldstad W.T. Hamlen. "Application Program Maintenance Study: Report to Respondents" Proceedings GUIDE 48, Philadelphia, PA, 1983

[21] [FOUR04] D. Fourney "Hierarchical program visualisation" 2004

[22] [FURN86] Furnas G, " generalized Fisheye Views", *Proceedings SIGCHI Human Factor in Computing*, 1986, pp 18-23.

[23] [GAMM95] E. Gamma, R. Helm, R. Johnson, J. Vlissides "Design Patterns – Elements of Reusable Object-Oriented Software", Addison-Wesley, 1994

[24] [GARG98] S. Garg, A. van Moorsel, K. Vaidyanathan and K. Trivedi. A Methodology for Detection and Estimation of Software Aging. In *Proc. ISSRE-98*, Paderborn, Germany.

[25] [GUPT97] Gupta R., Soffa M. and Howard J., "Hybrid Slicing: Integrating Dynamic Information with Static Analysis", *Trans. on SE and Methodology*, 6(4), pp 370-397, 10.1997.

[26] [HARM01] Harman M., Hierons R. M., Danicic S., Laurence M., Howroyd J. and Fox C, 2001, "Pre/Post Conditioned Slicing", *IEEE International Conference on Software Maintenance (ICSM'2001)*, Florence, Italy.

[27] [HEUZ03] D. Heuzeroth, T. Holl, G. Högström, W. Löwe "Automatic Design Pattern Detection" University of Karlsruhe,Germany IPD, Program Structures Group University of Växjö, Sweden MSI, Software Technology Group, 2003

[28] [HOPK01] Hopkins, J. and Fishwick, P. A., "A Three-Dimensional Human Agent Metaphor for Modeling and Simulation", *Proc. IEEE*, 89(2), 2001, pp 131-147.

[29] [HORW90] Horwitz S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs", ACM *Tr. on Progr. Lang. and Systems*, 12(1), pp. 26-61, 1990.

[30] [HUAN95] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, "Software Rejuvenation: Analysis, Module and Applications," *Proceedings of the 25th Symposium on Fault Tolerant Computer Systems*, Pasadena, CA, June 1995, pp. 381–390.

[31] [JERD94] Dean F Jerding and John T Stasko. Using visualization to foster object-oriented program understanding. Technical report, Georgia Institute of Technology, July 1994.

[32] [KELL99] R. Keller, R. Schauer, S. Robitalille, and P. Page. Pattern- Based Reverse-Engineering of Design Components. Proceedings of the 21st ICSE, pages 226–235, May 1999.

[33] [KEOW00] L. Keown, Virtual 3D Worlds for Enhanced Software Visualization. 2000.

[34] [KNIG99] Knight C., Munro M., "Visualising Software - A Key Research Area", *Proc. of the Int. Conference on Software Maintenance*; ICSM'99, IEEE Press, 1999.

[35] [KORE88] Korel, B. and Laski, J., "Dynamic program slicing", *In. Process. Letters*, 29(3), pp. 155-163, Oct. 1988.

[36] [LAMP95] Lamping, J., Rao, R., and Pirolli, P, "A focus + context technique based on hyperbolic geometry for visualizing large hierarchies.", *Proceedings SIGCHI Human Factors in Computing Systems*. 1995, pp 401-408.

[37] [LANZ02] M. Lanza "CodeCrawler – Lessons Learned in Building a Software Visualizaiton Tool" University of Berne, Switzerland, 2002.

[38] [LETO86] S. Letovsky, "Cognitive process in program comprehension", Enpirical studies of programmers, Ablex publishing corporation, 1986.

[39] [LEWE01] C. Lewerentz, F. Simon"Metric-Based 3D Visualisation of Large Object-Oriented Programs" Software and systems Engineering group, Technical university Cottbus, 2001

[40] [LITT86] D. Littman, J. Pinto, S. Letovsky, E. Soloway, "Mental models and software maintenance". Enpirical studies of programmers, Ablex Publishing Corporation, 1986.

[41] [LUKE80] F.J. Lukey. "Understanding and Debugging Programs." International journal of Man-Machine Studies, 1980.

[42] [MACK90] J.D. Mackinlay, S.K. Card and G.G. Robertson, "Rapid Controlled Movement through Virtual 3D Workspace." SIGGRAPH '90 Conference Proceedings. Computer Graphics, 24, 4, 171-176. 1990.

[43] [MACK91] Mackinlay J, Robertson G, Card S, "The Perspective Wall: Detail and Context Smoothly Integrated", *Proc.s SIGCHI Human Factors in Computing*, 1991, pp 173-179

[44] [MALE01] Maletic, J.I., Leigh, J., Marcus, A., Dunlap, G., "Visualizing Object-Oriented Software in Virtual Reality", *Proceedings of the 9th International Workshop on Program Comprehension (IWPC 2001)*, 2001, pp. 26-35.

[45] [MAYR94] A. Mayrhauser, and A.M. Vans "Comprehension Processes During large Scale Maintenance" 1994

[46] [MAYR95] A. von Mayrhauser, A. Vans, "Program comprehension during software maintenance and evolution" IEEE Computers, 1995.

[47] [MAYR98] Mayrhauser A., A. M. Vans, "Program Understanding Behavior During Adaptation of Large Scale Software", *Proceedings of the 6th Intl. Workshop on Program Comprehension.*, IWPC '98, pp. 164-172, Italy, June 1998.

[48] [MCIN04] P. McIntosh, M. Hamilton, R, Van Schyndel "X3D-UML: Enabling Advanced UML Visualisation Through X3D" School of Computer Science and Information Technology RMIT University, Melbourne, Australia 2004

[49] [MICH02] Michaud J., Storey M.-A.D. and Muller H.A., "Programs, Integrating Information Sources for Visualizing Java", *ICSM'2002*,

[50] [NEIL98] Nielsen, J., "2D is Better Than 3D", *AlertBox,* http://useit.com/alerbox/981115.html, 1998.

[51] [NEIL97] Nielson G. M, Hagen, H. and Mueller H., "Scientific Visualization: Overviews, Methodologies, and Techniques", *IEEE Computer Society*, 1997.

[52] [NGAN02]A. Nganou, R. Al-Amad, S. Shi and X. Xian "Software Visualization 2D versus 3D"

[53] [NOBL03] J. Noble, R. Biddle "Patterns as sign" Computer science Victoria University of Wellington, New Zealand. 2003

[54] [NOIK94] E. Noik, "A Space of Presentation Emphasis Techniques for Visualizing Graphs." Proc. Graphics Interfaces'94, 225-233. 1994.

[55] [OBRI03] M. O'Brien "Software comprehension – A review and research direction" Department of computer science and Information system. University of Limerick, Ireland, 2003.

[56] [OGL] www.opengl.org

[57] [ORNB92] S.B. OrnBurn, S.Rugaber. "Reserve Engineering: Resolving conflicts between expected and actual software design." Proceedings of the conference on Software maintenance, 1992.

[58] [PENN87] N. Pennington, "Stimulus structures and mental representations in expert comprehension of computer programs", Cognitive Psychology, 1987.

[59] [PITT98] F. Pittarello and A. Celentano "A Multimodal Approach for Orientation and Navigation in 3D Scenes" 1998

[60] [PIZK03] M. Pizka, A. Bauer"A brief Top-Down and Bottom-Up philosophy on software evolution" Germany 2003

[61] [PRIC93a] Blaine A Price, Ronald M Baecker, and Ian S Small. A principled taxonomy of software visualization. *Jornal of Visual Languages and Computing*, 4(3):211{266, september 1993.

[62] [PRIC93b] B.A. Price, R.M. Baecker and I.S. Small. A principled taxonomy of software visualization. Journal of Visual Languages and Computing, 1994.

[63] [RAJL02] V. Rajlich and N. Wilde, "The Role of Concepts in Program Comprehension" Proceedings of IWPC 2002,

[64] [REIS94] Reiss S. P., Cruz I. S, "Practical Software Visualization" *CHI'94 Workshop on SV.*

[65] [RILL02] Rilling J. and Mudur S. P., "On the Use of Metaballs to Visually Map Source Code Structures and Analysis Results onto 3D Space", *Proc. IEEE WRCE 2002.*

[66] [RILL02b] Rilling J. and Seffah A.; "The CONCEPT Project - Applying Source Code Analysis to Reduce Information Complexity of Static and Dynamic Visualization Techniques", IEEE VISSOFT workshop , Paris, 2002

[67] [RILL05] J. Rilling, V.-L. Nguyen "Visualization of 3D UML Diagrams and 3D Design Pattern Grouping" 3rd International workshop on Visualizing Software for Understanding and Analysis" Budapest, Hungary 2005.

[68] [ROMA92] G Ruia-Catalin Roman and Kenneth C Cox. Program visualization: The art of mapping programs to pictures. In *Proceedings of the 14th International Conference on Softeware Engineering*, May 1992.

[69] [RUGA95a] S Rugaber. Program comprehension. Georgia Institute of technology. 1995

[70] [RUGA95b] S Rugaber, K. Stirewalt, L. Wills "The interleaving problem in program understanding." 2nd working conference on Reverse Engineering, Toronto, Canada, 1995.

[71] [SANL01] Sanlaville R., Favre J.M., Y. Ledru, "Helping Various Stakeholders to Understand a Very Large Software Product", *European Conference on Component-Based Software Engineering,* Sept. 2001.

[72] [SCHA96] D. Schaffer, Z. Zuo, S. Greenberg, L. Bartram, L. Dill, S. Dubs, and M. Roseman, "Navigating hierarchically clustered networks through fisheye and full zoom methods." ACM Transactions on CHI 3(2), 162 - 188. 1996.

[73] [SCHA98] Schauer R and. Keller R.K., "Pattern Visualization for Software Comprehension", In the Proc. of the 6th IWPC, Ischia, Italy, June 1998. IEEE Computer Society. pp 4-1

[74] [SHAU98] R. Shauer, R.K. Keller "Pattern visualisation for software comprehension" Departement IRO, Universite de Montreal, 1998

[75] [SHI04] Shi Sheng Hua "3D Visualization of design patterns for large program comprehension" Concordia University, Canada, 2004

[76] [SHNE80] B. Shneiderman, Software psychology: Human factors in computer and information systems, Winthrop Publishers Inc, 1980.

[77] [SHNE92] Shneiderman, B., "Tree Visualization with Tree-Maps: A 2-D Space-Filling Approach". *ACM Trans. of CHI*, vol. 11, no.1, 1992, pp. 92-99.

[78] [SMIT03] J. Smith, D. Stotts "SPQR: Flexible Automated Design Pattern Extraction from Source Code" Technical report, department of Computer science. University of North Carolina, May 2003

[79] [STAP99] M.L. Staples and J.M. Bieman, "3D Visualization of Software Structure" Advances in Computers, Volume 49, Academic Press, London, 1999.

[80] [STOR97] Storey, M.-A.D.; Wong, K.; Muller, H.A., "How do program understanding tools affect how programmers understand programs?" Proceedings of the Fourth Working Conference on Reverse Engineering, Amsterdam, Netherlands, 6-8 Oct. 1997. pp. 12-21

[81] [STOR99] Storey M.-A., Fracchia F. and Müller H.., "Cognitive Design Elements to support the Construction of a Mental Model During Software Exploration, *Journal of Software Systems*, special issue on Program Comprehension, v 44, 1999, pp.171-185

[82] [SOLO84] E. Soloway, K.Ehrlich "Empirical studies of programming knowledge." IEEE Transactions on Software engineering, 1984.

[83] [STOR99] Storey M.-A., Fracchia F. and Müller H.., "Cognitive Design Elements to support the Construction of a Mental Model During Software Exploration", *J. of Softw. Systems*, Spec. Issue on P. Comprehension, v 44, pp.171-185, 1999.

[84] [THAD03] U. Thaden, F. Steimann "Animated UML as a 3D0illustration for teaching OOP" Learning lab lower Saxony, Hannover, 2003[VLIS98] J. Vlissides. "Notation, Notation, Notation. C++ Report", April 1998.

[85] [V_ASSIST] Visual Assist, Intelli-sense add-on to Microsoft Visual Studio www.wholetomato.com

[86] [WAIT84] W.M. Waite, G.Goos, "Compiler construction," Springer, N.Y. 1984.

[87] [WEIS82] Weiser M., "Program slicing", *IEEE Transactions on Software Eng., SE-10, No. 4*, 1982, pp. 352-35

[88] [WEIS84] M. D. Weiser: *Program Slicing*. IEEE Transactions on Software Engineering, 10, July 1984.

[89] [UML] http://www.uml.org/

[90] [YOUNG96] P. Young, "Software Visualization" Visualization Research Group Center for Software Maintenance, University of Durham, 1996.

[91] [ZHAN04] Y. Zhang "Software comprehension – A description logic approach" Concordia University, Canada, 2004