NOTE TO USERS

This reproduction is the best copy available.

UMI®

3D Visualization of Design Patterns for Large Program Comprehension

Sheng hua Shi

A Thesis

in

the Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

April 2004

© Sheng hua Shi, 2004

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisisitons et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-91113-6
Our file Notre référence
ISBN: 0-612-91113-6

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this dissertation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de ce manuscrit.

While these forms may be included in the document page count, their removal does not represent any loss of content from the dissertation.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canadä

Abstract

3D Visualization of Design Patterns for Large Program Comprehension

Sheng hua Shi

The advantages of object-oriented (OO) and other modern software engineering techniques are offset by the continuing increase in the size and complexity of software systems. Projects involving millions of lines of code are no longer exceptional. Such systems are generally extremely difficult to comprehend. Software Visualization (SV) techniques can play a significant role in the process of comprehending such systems. Numerous SV tools have been developed to support program comprehension during the past two decades; most of them produce two dimensional visuals. For large programs, the number of entities and their inter-relationships may be far too many for the tools to automatically produce a comprehensible 2D visual. With rapid advances in processing power and computer graphics techniques, three-dimensional visualization is gaining increasing attention in this domain. 3D graphics provides the extra 3rd dimension and increases the virtual space available for visual depiction of software entities and relationships.

Our research explores a new category of Software Visualization tools that enable users to visualize program analysis results in a three-dimensional virtual environment. We combine program analysis and software visualization techniques using a cityscape metaphor to present views of structures in software thus enhancing program understanding during software maintenance. We have developed and implemented the Java3D Virtual City (JVC) extensible framework to facilitate such exploration. Use of JVC in experiments on visualization of automatically discovered design patterns illustrates that the proposed methodology does indeed provide additional insights into complex relationships among data without having to analyze the underlying source code in detail. The framework includes substantial built-in functionality to automatically generate scene graph for easing understanding, and can be extended with other understanding-promoting techniques.

Acknowledgements

It would be impossible to have this thesis without support from my family and friends, my supervisors, and my colleagues. My thanks go to all those who have helped me during the time I work on the thesis research.

My wife, Han Li, provided me full support throughout the duration of my studies.

My thanks also go to Dr. Juergen Rilling, who has been a very nice supervisor. His guidance to this research is invaluable, and I have learnt a lot from him.

I wish to thank my co-supervisor, Dr. Sudhir Pandurang Mudur, who has been always being counted on when I encountered difficulties during the research. I have a lot of respect for him.

Thanks must also go to Man Li for her support over different things at different times during my studies and David Lu for his proof reading this document.

Many other members of the CONCEPT research team have provided me discussion, help, and friendships. I wish to express my special thanks to Wenjun Meng and Yonggang Zhang among these team members.

Table of Contents

List of Figures x

List of Tables xiii

Chapter 1 Introduction 1

1.1 Research Question 4

1.2 Contributions 5

1.3 Thesis Organization 6

Chapter 2 Program Comprehension 7

2.1 Program Comprehension 7

2.2 Program Analysis Techniques 12

2.2.1. Low Level Analysis 13

2.2.2. High Level Analysis 16

2.3 Program Understanding Tools 21

2.3.1. Commercial Products 24

2.3.2. Research Prototypes 27

Chapter 3 Software Visualization 31

3.1 Visualization 31

3.2 Software Visualization... 32

3.3 3D Software Visualization 35

3.3.1. 3D versus 2D Visualization 35

3.3.2. Important Issues 37

3.3.3. Advantages of 3D Visualization 41

3.3.4. Challenges of 3D Visualization 42

3.3.5. 3D Visualization Techniques 43

3.3.6. VR and 3D Visualization 51

3.4 Potential 3D Graphic Engines 52

3.4.1. OpenGL.. 53

3.4.2. Direct3D 56

3.4.3. Java3D 57

3.4.4. VRML 61

3.5 XML in the Visualization Pipeline 63

3.6 Selecting the Appropriate Graphic Engine 65

3.7 Related Work in 3D Software Visualization 66

3.7.1. Software World 66

3.7.2. sv3D 67

3.7.3. VizzAnalyzer 69

3.7.4. MetaViz 71

3.7.5. UML3D 72

Chapter 4 The Java3D Virtual City (JVC) 74

4.1 The Metaphor: the Virtual City 74

4.1.1. Why Virtual City? 74

4.1.2. The Virtual World Metaphor 75

4.1.3. The OO Program Artifacts 80

4.1.4. Mapping the Virtual City Metaphor to OO Artifacts 82

4.2 System Overview 83

4.2.1. Proposed Approach 84

4.2.2. The CONCEPT Project 86

4.2.3. The Visualization Process 87

4.2.4. The GUI Model 88

4.2.5. The 3D Graphics Model 90

4.2.6. The Visualization Model 93

4.3 Implementation Issues 96

4.3.1. The System Structure of JVC 96

4.3.2. Layout and Grouping 98

4.3.3. Navigation Methods 103

4.3.4. Memory Usage 108

Chapter 5 Application of JVC: Visualizing Design Pattern 109

5.1 Introduction to Design Patterns 110

5.2 Design Pattern Recovery 112

5.2.1. Literature Review 112

5.2.2. Design Pattern Recovery in CONCEPT 114

5.3 Visualizing Design patterns 116

5.3.1. Target System 116

5.3.2. The Input XML File 116

5.3.3. The Results 117

5.3.4. Discussions and Limitations 123

Chapter 6 Conclusions and Future Works 124

References 127

List of Figures

Figure 1: Main window of Imagix 4D 25

Figure 2: Cross Referencer in SNiFF+ version 4 26

Figure 3: Multiple views in SeeSoft 27

Figure 4: An Example of Hierarchical View in SHriMP 29

Figure 5: A Rigi view 30

Figure 6: Mapping 2D sequence diagram notation into 3D space 37

Figure 7: 3D surface plots using Java 3D 44

Figure 8: Cityscape for software visualization 45

Figure 9: A class schedule on perspective walls 47

Figure 10: View of a file structure using Cam trees 48

Figure 11: A user immersed in the VR 52

Figure 12: Block diagram of the OpenGL pipeline 54

Figure 13: Direct3D Graphics Pipeline 57

Figure 14: Java 3D application scene graph. 59

Figure 16: A VRML Virtual City in Cortona Browser 62

Figure 17: A VRML Campus Browsed with Cortona VRML Client 62

Figure 18: An overview of a district in Software World 66

Figure 19: Overview of a system in sv3D 68

Figure 20: View of one file in sv3D 68

Figure 21: A package hierarchy using the fully qualified class names............................ 69

Figure 22 Top view of a 3D city with business information 70

Figure 23: A view of CBO in MetaViz.. 71

Figure 24: A class diagram of a system with over 200 classes.. 72

Figure 25: An example of a Virtual City 85

Figure 26: The Architecture of CONCEPT 86

Figure 27: The Visualization Process in JVC 87

Figure 28: The main window of JVC 88

Figure 29: Class diagram for a typical scene graph in JVC... 92

Figure 30: The detailed dataflow within the CONCEPT 93

Figure 31: JVC Visualization Model 94

Figure 32: The XML pipeline in JVC... 95

Figure 33 : The class diagram of JVC data model 96

Figure 34 Class diagram for JVC system structure 97

Figure 35: Control flow of Hill Climbing in JVC.. 101

Figure 36: Group and split cells in a grid 103

Figure 37: A global view 104

Figure 38: A detailed view.. 105

Figure 39: Navigate in JVC using data glove 108

Figure 40: Sample XML data in JVC 117

Figure 41: Design pattern visualization in JVC.. 118

Figure 42: Close view of a design pattern 119

Figure 43: Java Doc of a class 119

Figure 44: Detailed view of a design pattern 120

Figure 45: The ComboBox for selecting options.. 121

Figure 46: A virtual city shows pattern classes only (top) or classes without the pattern classes (bottom) 122

List of Tables

Table 1 Low level analysis resulting information categorization 13

Table 2 List of program understanding tools 23

Table 3: Comparison of 3D APIs 65

Table 4 Language issues of OO programs 81

Table 5 Mappings 83

Table 6 Design pattern space 112

Chapter 1

Introduction

Software visualization has been used for various purposes during the past two decades [STAS92, PRIC93, GOME 01, RILL03]. Some visualization tools show animations of algorithms and data structures [STOR97], often for educational purposes. Others show the run-time behavior of a program to aid debugging, testing, or performance optimization tasks [STOR97].

For large, complex software systems, the comprehension of such diagrammatic depictions is restricted by the resolution limits of the visual medium (2D computer screen) and the limits of user's cognitive and perceptual capacities. One approach to overcome or reduce the limitations of the visual medium is to make use of a third dimension by mapping source code structures and program executions to a 3D space [CHAR02]. Mapping these program artifacts into the 3D space allows users to identify common shapes or common configurations that may become apparent, and which could then be related directly to design features in the code.

Program comprehension can be described as the process of analyzing a subject system (a) to identify the system's components and their interrelationships, (b) to be able to create representations of the system in other forms at higher levels of abstraction and (c) to understand the program execution and the sequence in which it occurred. Software visualization techniques can assist considerably in this process by creating visual

representations that bring out complex relationships and make it easier to comprehend them in comparison to textual representations. It would be ideal to be able to simultaneously view and understand detailed information about a specific activity in a global context at all times for any size of program. As Ben Shneiderman explains in [SHNE 94], the main goal of every visualization technique is: "Overview first, zoom and filter, then details on demand". This means that visualization should first provide an overview of the whole data set then let the user restrict the set of data on which the visualization is applied, and finally provide more details on the part the user is interested in. Software visualization of source code can be further categorized in static views and dynamic views. The static views are based on a static analysis of the source code and its associated information and provide a more generic high-level view of the system and its source code. The dynamic view is based on information from the analysis of recorded or monitored program execution. Based on their available run-time information, dynamic views can provide a more detailed and insightful view of the system with respect to a particular program execution. As Mayrhauser [MAYR97] illustrated, dynamic and static views should be regarded as complementary views rather than being mutually exclusive.

This thesis explores a new category of Software Visualization tools that enable a user to graphically visualize program analysis results in a three-dimensional virtual environment. We combine program analysis and software visualization techniques within a virtual city environment to present views of structures in software thus enhancing program understanding during software maintenance.

The objective of this research is to develop and explore the use of a cityscape metaphor for visualizing a system's structural information obtained through program analysis techniques. Towards this objective we have developed and implemented the Java3D Virtual City (JVC) framework that can facilitate this exploration. The framework includes substantial built-in functionality to automatically generate Java 3D scene graphs that can be easily rendered for easing understanding. JVC architectural design is such that it can be easily extended with other techniques for promoting program understanding.

Using this JVC framework we have carried out a number of experiments in visualizing the results of automatic design pattern analysis of large programs. These experiments illustrate that the methodology proposed in this research and provide additional insights into 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 explicit in the software documentation or source code.

In summary, our research focuses on the area of particular interest determined by the following parameters:

- Visualization of object-oriented systems
- Utilization of 3D space to take advantage of additional navigation and visualization techniques.
- Support for visualization of both static and dynamic analysis information.

1.1 Research Question

The goal of a person who comprehends a software system is to build progressively refined mental models of the system [STOR99]. The mental models proposed by various researchers in the field of program comprehension have a commonality in that they all are composed from semantic constructs. These constructs are typically abstractions, at various levels, of program features. The network formed from these constructs constitutes the maintainers understanding and representation of the program. Software visualization attempts to provide a mapping from the program code to a visual representation that matches the maintainer's mental model as closely as possible. Software, despite many software engineering advances in requirements, design, and implementation techniques, continues to be large and complex, providing additional challenges with respect to scalability, navigation and applicability for various comprehension tasks. These lead to our research question:

How to visualize source code structures?

We argue that the necessary information resides in a software system at various levels of abstraction. In this thesis, we identify three levels of abstraction: system / architecture level (information about the whole system and its overall structure), class level (information about the structure of classes and class hierarchies), and source level (information about the target code itself and related documentation). This approach is closely related to the survey result by Rainer Koschke [KOSC03], which concludes that multiple views are needed in according to information at different levels of granularity: source level, middle level, architecture level. Introducing these different levels of

granularity allows us to break down the research question in several more distinct aspects, such as:

- How are these multiple views organized and connected?
- How can one navigate within and between views?
- How can the visualization maintain the context during navigation?
- Can the visualization answer these questions intuitively:
  - How big is the software system and how is it structured?
  - What is the architecture of the system and what are the subsystems?
  - Where are the most important classes and class hierarchies that represent the problem domain and make the whole software work?
  - Where have design patterns been used and which ones?

1.2 Contributions

The contributions of this thesis can be summarized as:

1. An exploratory study of advantages of 3D techniques and displaying information in a new and different form for large program comprehension.

2. Definition of a virtual city metaphor and the required mappings and representations for the purpose of visualizing software as an aid to program comprehension.

3. An implementation that combines Java, Java3D, XML, VRML, and other visualization techniques resulting in the Java3D Virtual City (JVC) extensible framework.

4. Experiments on design pattern visualization that illustrate the possibility of providing additional insight of a system using JVC.

1.3 Thesis Organization

The thesis is organized as follows. Chapter 2 introduces background information relevant to program comprehension, program analysis techniques, and program understanding tools. Chapter 3 discusses software visualization and related issues, with a focus on 3D visualization techniques, and related work in 3D software visualization. Chapter 4 introduces the Java3D Virtual City project and describes its system structure and some of the major design and implantation issues. Chapter 5 provides a case study, applying the JVC project to visualize design patterns that were recovered from existing source code. Chapter 6 concludes with a summary of the contributions of this research and discusses future work.

Chapter 2

Program Comprehension

This chapter gives an overview of *Program Comprehension, Program Analysis* techniques, and some selected program understanding tools in order to establish the background context for the visualization work carried out in this thesis.

2.1 Program Comprehension

System design and its comprehension are two of the most important tasks in the software development and maintenance cycle, defining how the different system parts are organized and communicate with each other. Large software systems place an enormous cognitive load on users [RILL02]. They have often a large number of components along with complex interactions amongst them. In traditional debugging and programming environments the primary technique for visualizing programs is by displaying source code lines associated with a particular program, providing very limited support of visual abstractions or guidance during program comprehension. On the other hand, modern program environments attempt to enhance the functionality of traditional debuggers by providing information about class names (and their associated member functions and data members) that were either declared or included within the program context. In spite of this, a programmer will still have to observe large amounts of data without having any global view of the system. Advances in hardware and software, particularly high-density bit-mapped monitors and window-based user interfaces, have led to a renewed interest in

graphical representation of software [BALL96, MALE01, RILL03]. Visualization facilitates discovery by revealing hidden structures and behaviors in model output. It is in the areas of insight and understanding that visualization plays a central role [SHNE92, WALK98]. Graphical representations have been recognized as having an important impact in communication from the perspective of both writers and readers [MAYR98].

Program comprehension is a crucial part of system development and software maintenance. It is expected a major share of system development effort goes into modifying and extending existing systems. However, comprehending software systems, especially large legacy systems, is technically difficult, because they typically suffer from several problems, such as developers no longer available, outdated development methods that have been used to write the software, outdated or completely missing documentation, and, in general, a progressive degradation of design and quality. Rugaber summarized that there exists five gaps in program comprehension [RUGA96]:

- The gap between application domain and programming languages. A programming language is just a model environment to solve some real problem. While tools exist to assist in understanding what the code is doing from a code perspective, there is little to assist the reverse engineer in determining, from a domain perspective, what is occurring with the code.

- The gap between the physical code and the abstract of high level design. Abstract concepts quickly become lost in the minutia detail of programming.

- The gap between originally structured system and the actual system with structure disintegrated. Even when good documentation is available for a system, maintenance over time causes the structure to drift from the original specification.

- The gap between the hierarchical world of programs and the associational nature of human cognition. Computer programs are formal, hierarchical expressions. Humans think in associative "chunks" of data. A reverse engineer must be able to build up correct high level chunks from the low level details evident in the program.

- The gap between the bottom-up source code analysis and the top-down application analysis. Code analysis is by its nature a bottom-up exercise. It requires, simultaneously, higher level meaning to be extracted from code fragments, and higher level concepts to be mapped to lower level implementations.

Moreover, the increasing size and complexity of software systems introduce new challenges in comprehending the overall program structure, their artifacts, and the behavioral relationships among these artifacts.

Numerous theories have been formulated and empirical studies have been conducted to achieve different comprehension strategies and methods [STOR97b]. Though different programmer might adopt different strategy in program comprehension due to their characteristics, differences among programs, and aspects of the task at hand, program understanding tools should facilitate the comprehension strategies used by programmers to achieve specific maintenance tasks. For this reason, we give a brief description to several cognitive models.

Bottom-up

Shneiderman [SHNE80] proposed that programs are understood bottom-up, by reading source code and then mentally chunking low-level software artifacts into meaningful, higher-level abstractions. These abstractions are further grouped until a high-level understanding of the program is formed.

Pennington [PENN87] also observed programmers using a bottom-up strategy initially gathering statement and control- flow information. These micro-structures were chunked and cross-referenced by macro-structures to form a program model. A subsequent situation model was formed, also bottom-up, using application-domain knowledge to produce a hierarchy of functional abstractions.

Top-down

Brooks [BROO83] suggested that programs are understood top-down, by reconstructing knowledge about the application domain and mapping that to the source code. This strategy begins with a global hypothesis about the program. This initial hypothesis is refined into a hierarchy of secondary hypotheses. Verifying or rejecting a hypothesis depends heavily on the presence or absence of beacons (cues). Soloway and Ehrlich [SOLO84] observed that a top-down strategy is used when the program or type of program is familiar. They also observed that expert programmers recognized program plans and exploited programming conventions during comprehension.

Knowledge-based

Letovsky [LETO86] theorized that programmers are opportunistic processors capable of exploiting either bottom-up or top-down cues. This theory has three

components: a knowledge base that encodes the programmer's application and programming expertise; a mental model that represents the programmer's current understanding of the program; and an assimilation process that describes how the mental model evolves using the programmer's knowledge base and program information. Inquiry episodes are a key part of the assimilation process. Such an episode consists of a programmer asking a question, conjecturing an answer, and then searching through the code and documentation to verify or reject the conjecture.

Systematic and as-needed

Littman et al. [LITT86] observed that programmers use either a systematic approach, reading the code in detail and tracing through control and data flow, or an as-needed approach, focusing only on the code related to the task at hand. Soloway et al. combined these two theories as macro-strategies aimed to understand the software at a more global level. In the systematic macro-strategy, the programmer traces the flow of the whole program and performs simulations as all of the code and documentation is read. However, this strategy is less feasible for larger programs. In the more commonly used as-needed macro-strategy, programmers look at only what they think is relevant. However, more mistakes could occur since important interactions might be overlooked.

Integrated approaches

Von Mayrhauser and Vans [MAYR95] combined the top-down, bottom-up, and knowledge-based approaches into a single metamodel. They proposed that understanding is built concurrently at several levels of abstractions by freely switching between the three comprehension strategies. Von Mayrhauser and Vans [MAYR95] combined

Soloway's top-down model with Pennington's program and situation models. In their experiments, they observed that some programmers frequently switched between all three of these models. They formulated an integrated metamodel where understanding is built concurrently at several levels of abstractions by freely switching between the three comprehension strategies.

2.2 Program Analysis Techniques

Software visualization allows for the transformation of a large amount of data to a higher level of abstraction that improves the comprehension of the overall program structure. However, even with higher levels of visual abstractions, a user might still have to deal with a large amount of data without having any meaningful insights about the relationships and the dependencies within a given scenario. Filtering and interpreting enormous quantities of information is a problem for humans. From a mass of data they need to extract knowledge that will allow them to make informed decisions. The algorithmic support addresses these limitations of the general visualization techniques.

Program analysis denotes the initial step towards program understanding. The use of reverse engineering and analyzing existing program executions allows us to derive data and control dependencies that can be used for various algorithmic approaches to provide users with additional insight in dependencies and relationship among program artifacts that exist in a program and its executions. Typically, a program performs a large set of functions/outputs. Rather than trying to comprehend all of a system's functionalities, programmers tend to focus on selected functions (outputs) and those parts of a program that are directly related to that function. There exist numerous analysis techniques that

retrieve program information reliably from source code and establish appropriate models for further investigation and modification [NIEL99]. Welf et al. suggested distinguish these techniques between two kinds: low and high levels [WELF02].

2.2.1. Low Level Analysis

The low level analyses can be grouped into the following two categories: static and dynamic analysis with the static analysis to focus on structural and dynamic analysis on behavioral aspects (see Table 1).

Table 1 Low level analysis resulting information categorization

	Structural	Behavioral
Static	Syntactic and type information.	Information such as the actual types, call graphs, and control flow graphs associated with the system
Dynamic	Information based on program executions.	Information assembled during single or multiple program executions.

In what follows, we present a general overview of different analysis techniques, their correlation with other high-level analyses, and their representations.

Lexical Analysis

The objective of the lexical analyzer, frequently referred to in the literature as a scanner, lexer, tokenizer or linear analyzer [WAIT84], is to look for patterns in a source program in order to convert them to a sequence of symbols, so-called tokens. The lexical analyzer provides structural program information as results. For example, *Agrep* is a UNIX utility that allows for approximate matching between a regular-like pattern and text in source code or plain text files [WU92].

Syntax Analysis

The syntactical analyzer, also known as a parser, structural or hierarchical analysis, checks the order of the tokens provided by the scanner against some syntactical rules. The hierarchical analysis usually results in a parse tree, Abstract Syntax Tree (AST), Directed Acyclic Graph (DAG) or any other syntactical structure [WAIT84, TOGE01, LUDW01]. The syntactical analyzer provides static structural program information as results.

Semantic Analysis

The semantic analysis is a context-sensitive analysis. It is performed in order to check the semantics of a source program [WAIT84, TOGE01, LUDW01]. To ensure the correct semantics of a program, semantic rules are attached to the lexical and syntactical productions. The semantic analyzer is therefore mostly implemented as a part of the syntactical analyzer. The semantic analyzer outputs static structural program information.

Dataflow Analysis

Data flow analysis (DFA) is a process for collecting problem specific behavioral information about programs, such as the use, definition, and dependencies of data, without actually executing the program [NIEL99]. It is impossible to determine the exact output of a program since it cannot be determined which execution path in the control flow graph is actually taken. This means that all possible paths are considered to be actual execution paths. The result of this assumption is that one can only obtain approximate solutions for certain data flow problems [MARL90].

Program Slicing

Weiser's original research [WEIS84] on program slicing was motivated by the need to help students understand and debug their programs. Weiser discovered that the mental process used by programmers when debugging their code was slicing and tried to formally define this process and the output by introducing his first algorithm.

Weiser [WEIS84] defined a static program slice as those parts of a program P that potentially could affect the value of a variable v at a point of interest. A large number of extensions to the original slicing algorithms have been presented in the literature, e.g. [AGRA90, AGRA93, HARM98, HARM01, HORW90, HORW92, LARS96].

Korel and Laski [KORE97] introduced the notion of dynamic slicing that can be seen as a refinement of the static approach by utilizing additional information derived from program executions on some specific program input. The dynamic slice preserves the program behavior for a specific input, in contrast to the static approach, which preserves the program behavior for the set of all inputs for which a program terminates. Later extensions of the dynamic slicing algorithms include hybrid algorithms [GOPA91, KAMK95, KREU99, ZHAO98] that combine static and dynamic information for the slice computation were introduced in [GUPT97, RILL01].

The reason for this diversity of slicing techniques and criteria is that different applications require different properties of slices. Program slicing is not only used in software debugging [AGRA93, GUPT97, KORE97, LYLE86, RILL02, TIPF95], but also in software maintenance and testing [MART94, HORW90, RILL01]. There are

two major approaches to locating places of low design quality in an existing design. The first one is a systematic approach that requires a complete understanding of the program behavior before any code modification. The second one is an "as needed" approach that can be adopted as it requires only a partial understanding of the program to locate as quickly as possible certain code segments that need to be changed, tested, or maintained for desired enhancement or bug-fixing.

2.2.2. High Level Analysis

Low-level analyses usually result in representations that are much too complex to provide the engineer with any architectural insight. To achieve an architectural understanding, we must therefore reduce the complexity of the low-level representation by means of high level analyses. High-level analysis uses different techniques to focus on higher level abstracts. Typical examples for such higher-level analysis approaches are the identification of design patterns, individual components, connectors, or architectural styles. In what follows, we describe some of these techniques.

Software Metrics

The term software metrics [COTE88] is not uniquely defined. In literature, software measure, software measurement, and software metrics are often used interchangeably.

While the use of metrics is commonplace in the traditional engineering world, it has yet to become fully established in the software domain, even though it has been shown that software metrics can provide software engineers and maintainers with

guidance in analyzing the quality of their design and code and its future maintainability [BASI95, BASI96, FENT91, LIB01, LIW93, MART94]. Software metrics could address many aspects of the software life cycle - process, product, people, quality, design maintenance etc. Li and Henry [LIW93] concluded after using their metrics to evaluate two software systems that there “is a strong relationship between metrics and maintenance effort in object-oriented systems” and that “Maintenance effort can be predicted from combinations of metrics collected from source code”.

Software design metrics

Several design metrics are presented in the literature [FENT91, HITZ96, LIB01, MART94] to evaluate the design and quality of software systems, enabling the early identification of maintenance and reuse issues in existing systems. Two major categories of design metrics can be distinguished: cohesion (internal aspects) and coupling (structural aspects) design metrics. Metrics measure certain properties of a software project by mapping them to numbers, according to well-defined, objective measurement rules. The measurement rules are then used to describe, judge, or predict characteristics of the software project with respect to the property that has been measured. Usually, measurements are made to provide information with which better decisions about software engineering tasks can be planned and carried out.

Variants of metrics are e.g. complexity metrics, which measure the complexity of an entity of a system; coupling metrics, which measure the coupling between classes; cohesion metrics, which measure the closeness of entities of a class and inheritance;

and tree metrics, which try to retrieve the program's OO concept to depict relationships between objects.

Component and Communication Detection

Large systems are divided into subsystems. These subsystems, also known as components, and the dependencies that exist among the components form the different layers within software architecture. Different techniques based on *metrics*, *dominance analysis* or *concept analysis*, as well as *design pattern detection* can be found in [KOSS00]. This includes automatic techniques, such as *connection-based, metric-based, slicing-based, graph-based, concept-based,* and *DFA-based* approaches.

Design Pattern Detection

Design patterns [GAMM94] 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]. During forward engineering, design patterns are used as good standard solutions for implementing software with certain properties. Inversely, during reverse engineering, one can try to detect the patterns in the code in order to understand the intended high-level design. This is the approach most relevant for program comprehension. Design patterns are medium-scale patterns. They are smaller in scale than architectural patterns, but tend to be independent of a particular programming language or programming paradigm. Design pattern detection is needed for design recovery, comprehension, and documentation. Through the detection of design patterns, a system can be depicted on a higher level of abstraction. Further,

design patterns are potentially useful both in developing new designs and in comprehending existing OO designs. However, too often documentation shows endless detail about features, but fails to show how everything fits together, the fundamental architectural concepts.

Reverse engineering focuses on creating "representations of a system in another form at a higher level of abstraction" [CAMP97]. Many researchers have presented their efforts on automatic recovery of design patterns in large object oriented programs. As we shall discuss below, most of these adopt the approach of describing patterns at some level of formalism and then match the descriptions with the facts extracted from source code.

The Pat [KRAM96] system is a *Reverse Engineering* tool that searches for design pattern instances in existing software. Within Pat, design information is "extracted directly from C++ header files and stored in a repository" [KRAM96], "the patterns are expressed as PROLOG rules and the design information is then used to search for all patterns" [KRAM96]. In [ANTO98], Antoniol et al. present a more conservative approach to recover design patterns from design and source code. Their approach is mainly based on "a multi-stage reduction strategy using software metrics and structural properties to extract structural patterns for OO design or code" [ANTO98]. Similar to the Pat system, this approach also focuses on the same five structural patterns mentioned in [KRAM96]. In [FERE01], Ferenc et al. state that Columbus and Maisa pairs, two reverse engineering tools, can be used to document and analyze software implemented in C++ and to verify the architectural design decisions during

the software implementation phase as well. In [HEUZ03], Dirk Heuzeroth et al. argue that static analyses are not sufficient for pattern recovery, and they therefore introduce dynamic analysis techniques to detect design patterns in legacy code. Through their approach, a set of potential patterns are firstly detected by a static semantic analysis tool.

Dominance Analysis

Dominance Analysis [GIRA97] is a graph theoretic method that often has been applied to call graphs in imperative languages to identify so called dominant nodes (i.e. nodes that are the only entry point to a subsection of the graph). The dominant node and the subsection they dominate are then suitable module candidates since they represent a set of procedures with a low external coupling. This idea can easily be transferred to an object oriented setting by applying it to a graph of classes interacting with methods.

Concept Analysis

This general technique is often applied to software re-engineering. The concept lattice is computed on a relation of (abstract) objects and attributes. In the context of software understanding, this could be algorithm classes and data structure classes respectively. In feature analysis, e.g., these are use cases and classes, respectively. The analysis computes maximum sets of objects using the same set of attributes. There is an order relation defined on the maximum sets forming the concept lattice.

Architecture Style Detection

There is no unique definition of software architectures. Architectural patterns, which represent the highest-level patterns, are templates for concrete software architectures. They specify the system-wide structural properties of an application, and have an impact on the architecture of its subsystems. Reversing software architecture aims to: recover lost architectural information, update architecture documentation, support maintenance (comprehension) activities, provide different views on architecture, prepare for another platform and facilitate *impact analysis*.

No matter whether it is low or high level analysis, static or dynamic analysis, these analyses are not standalone by nature. System engineers have to be involved to accept or reject certain results proposed by the automatic analyses. The results of such analyses ought to be presented in a form that is intuitive to them. Therefore, program analysis must go hand in hand with tool support and interactive *Software Visualization*. Before we discuss software visualization in details, we give a brief review of existing tools.

2.3 Program Understanding Tools

Much effort has been put to develop tools aiming to aid the job of program understanding and maintenance. In order to examine the current status of the tool development, we briefly list some of them in Table 2 [SOUR 04]. It is impossible to provide a complete list because there are too many tools and it is beyond the purpose of this thesis to do so. More details can be found in [SOUR04].

We note that none of these tools is popularly used in industrial or academic environments. Most of them are basically code browsers with little advanced visualization technique support, and rarely have high-level program analysis functionality.

Table 2 List of program understanding tools

Tool	*License*	OS and Languages	Features
CC-Rider	*Commercial*	OS/2/Windows, C/C++	• Source browsing and documentation generating • API for customized software [WEST04]
Cflow	*Commercial*	Unix, C/C++/Lex/Yacc	• Analyzes source/object files and writes a chart of their external function references as a text-only diagram [CFLOW04]
CIAO	*Evaluation*	Solaris/Sun4/SGI C/C++/Java	• A graphical navigator for software and document repositories [ATT04]
CodeSurfer	*Commercial*	Solaris, C	• Program analysis and comprehension software • Has a dependence analyzer [GRAM04]
CodeWalker	*Commercial*	Solaris/Sun4/SGI C/C++/Java	• Graphical navigation tool to explore source code [GLOB04]
Cscope	*Commercial*	Unix, C/Lex/Yacc	• Quick search for definitions, declarations, usages of functions, variables and other symbols [CSCO04]
Ctags	*Commercial*	Unix, C	• Generates a tag file for the sources for quickly locating objects [CTAG04]
CXref:	*Free*	Unix, C,	• Generates documentation and cross-references from the source [CXRE04]
Source Browser	*Free*	Windows, C	• Simple program that displays the function call relationships in C programs [MORA04]
Source Insight	*Commercial*	Windows, C/C++/Java	• A source code editor enhanced for C/C++ and Java support • Multiple windows • Quick jump to the locations of defines, symbols, and functions etc. [DYNA04]
Source Navigator	*Commercial*	Unix/Windows, C/C++/Java/Tcl /Fortran/Cobol/ Assembly	• Enhanced source browsing, showing relationships (call/callby/include/includeby/etc.) between the various parts of the program [CYGN04]
Understand for C++	*Commercial*	Unix/Windows, C/C++	• A source code comprehension and documentation tool [SCIE04]

However, many existing tools are currently enriching their comprehension techniques by taking integrating the advances in software visualization, as necessary hardware and technology advances. As a result, software visualization tools, a subset of program understanding tool, are emerging to a more main stream technology to support the software comprehension process. Consequently, software visualization tools should adapt to and facilitate these different requirements. When the size of software becomes large, software visualization faces new challenges. In [MICH01], several reasons for these difficulties are mentioned: "(1) the diagram complexity is increased because of the large amount of information to be displayed, (2) the awkward layout techniques provided by the visualization approach, (3) their non-intuitive navigation, and (4) often their very specialized scope in depicting only certain program artifacts and their relationships." In what follows we selected two industrial strength and four research prototypes to illustrate on the one hand the state-of-the art software visualization tool support, as well as to provide an overview of the diversity among the tools and their supported visualization techniques.

2.3.1.Commercial Products

Imagix 4D

Imagix 4D [IMAG04] is a comprehensive program understanding tool, which enables the user to rapidly check or systematically study target software on any level -- from its high level architecture to the details of its build, class, and function dependencies. User can visually explore a wide range of aspects about the software - control structures, data usage, and inheritance. The user can review the software

metrics and source checks to identify and eliminate problem areas in the software. It can create design documentation automatically, further leveraging the information that Imagix 4D collects about the software. Figure 1 shows its main window. Its major features are:

- Data Collection and Visualization
- UML Diagrams and Other High Level Abstractions
- Flow Charts and Control Flow Analysis
- Software Metrics and Source Checks
- Automated Documentation

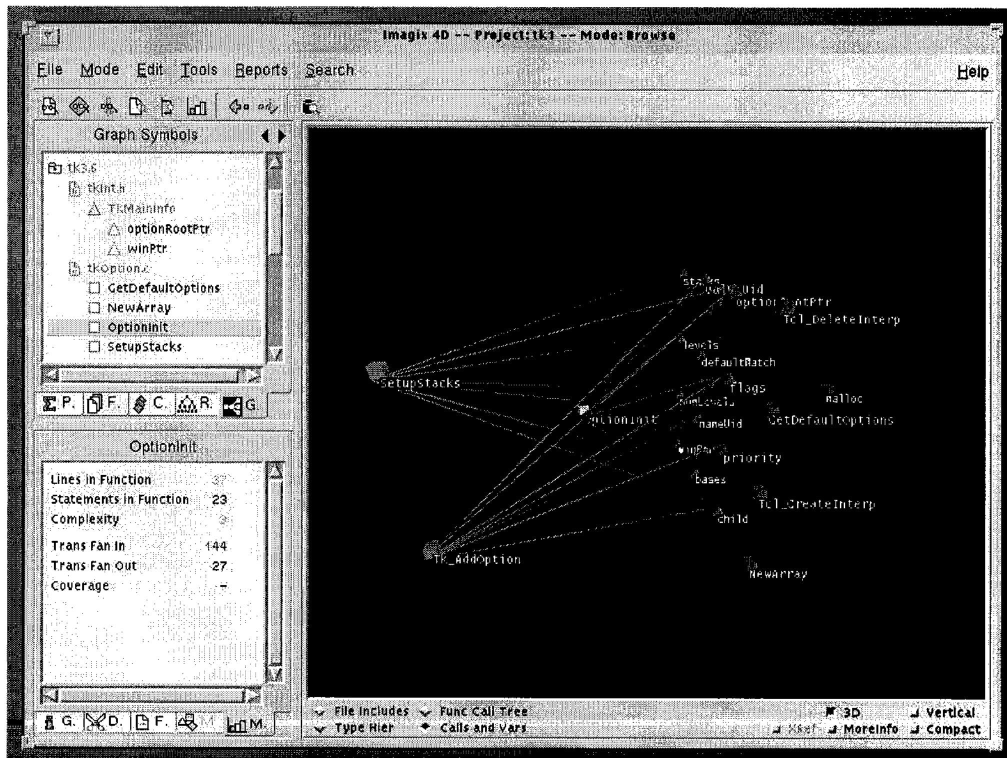

Figure 1: Main window of Imagix 4D

SNiFF+

SNiFF+ [SNIF04] is one of the better known commercial offerings, an integrated source code analysis and development environment for C, C++, Java, FORTRAN and CORBA IDL with large amounts of application code. SNiFF+ supports not only reverse engineering, but also configuration management, workspaces and build management. It provides the most comprehensive set of browsers and parsers. 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 at maximum efficiency. Figure 2 illustrates an example of displaying cross reference information in the tool.

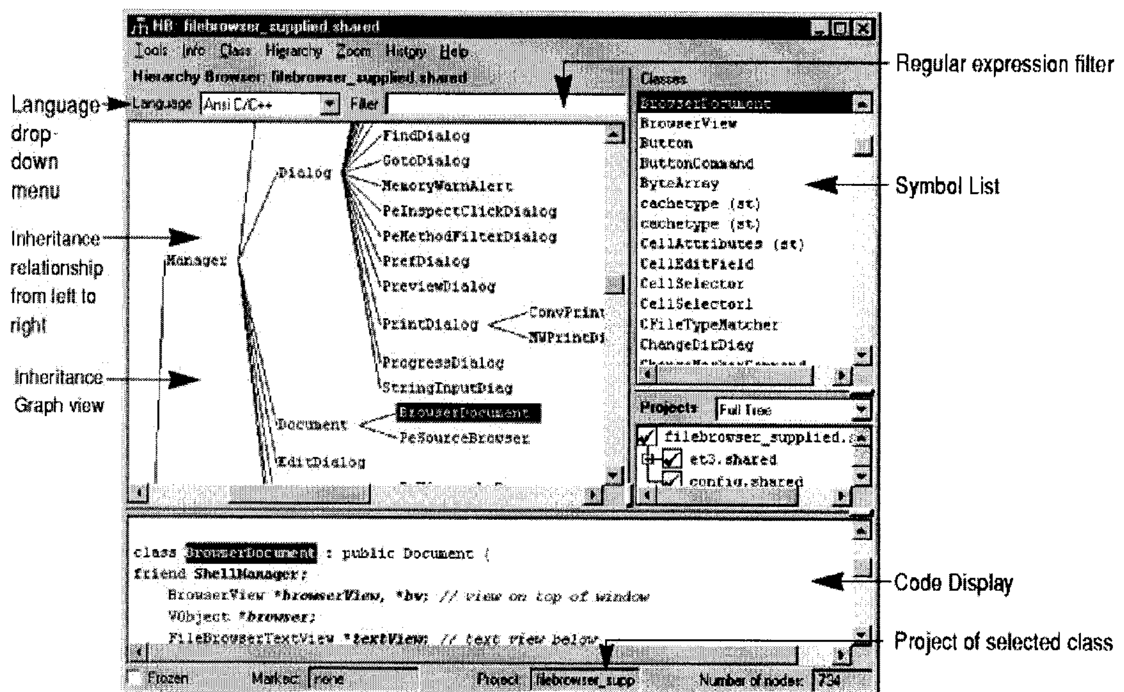

Figure 2: Cross Referencer in SNiFF+ version 4

2.3.2.Research Prototypes

SeeSoft

SeeSoft [EICK92] is a tool for visualizing software statistics from a variety of sources, including: version control systems, static analysis (call sites), dynamic analysis (profiling). The statistics are displayed using a color coded reduced representation, intended to allow patterns to be spotted easily. The visual representation allows the side-by-side comparison of a number of source files. Users could browse code in different levels of abstraction in SeeSoft and thus obtain overview and detailed information of source code. In Figure 3, for example, the color is used to represent the age of the code, e.g. blue represents old code and red represents the new.

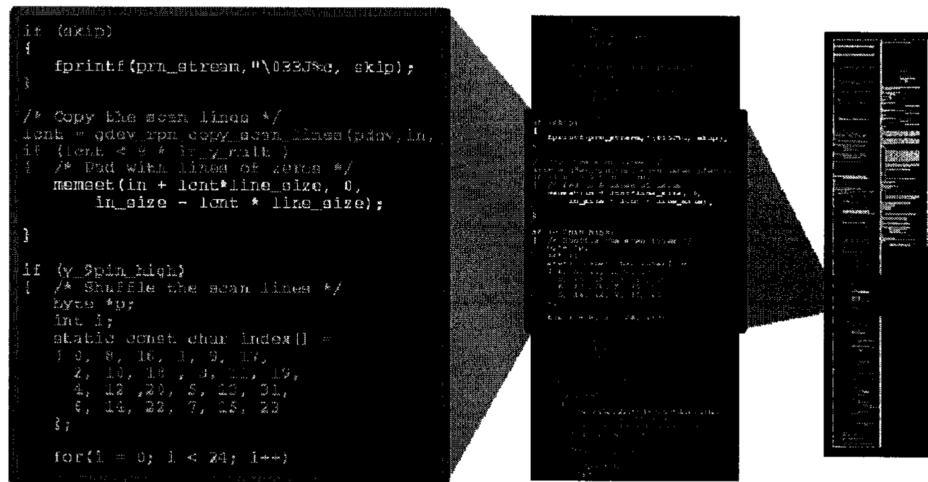

Figure 3: Multiple views in SeeSoft

SHriMP

The SHriMP [STOR01] (Simple Hierarchical Multi-Perspective) visualization technique was originally designed to enhance how programmers understand programs. This research tool presents a nested graph view of software architecture. Program source code and documentation are presented by embedding marked up text fragments within the nodes of the nested graph. Finer connections among these fragments are represented by a network that is navigated using a hypertext link-following metaphor. SHriMP combines this hypertext metaphor with animated panning and zooming motions over the nested graph to provide continuous orientation and contextual cues for the user. Its features include:

- Hierarchical view, overview of entire dataset (See example in Figure 4)
- Searching metrics
- Thumbnail view, a way of maintaining context while navigating
- Filmstrip for saving several different states of a project
- Tool tip
- Label
- Filtering
- Multiple layouts: Grid, Spring, and Tree
- Fisheye Zooming

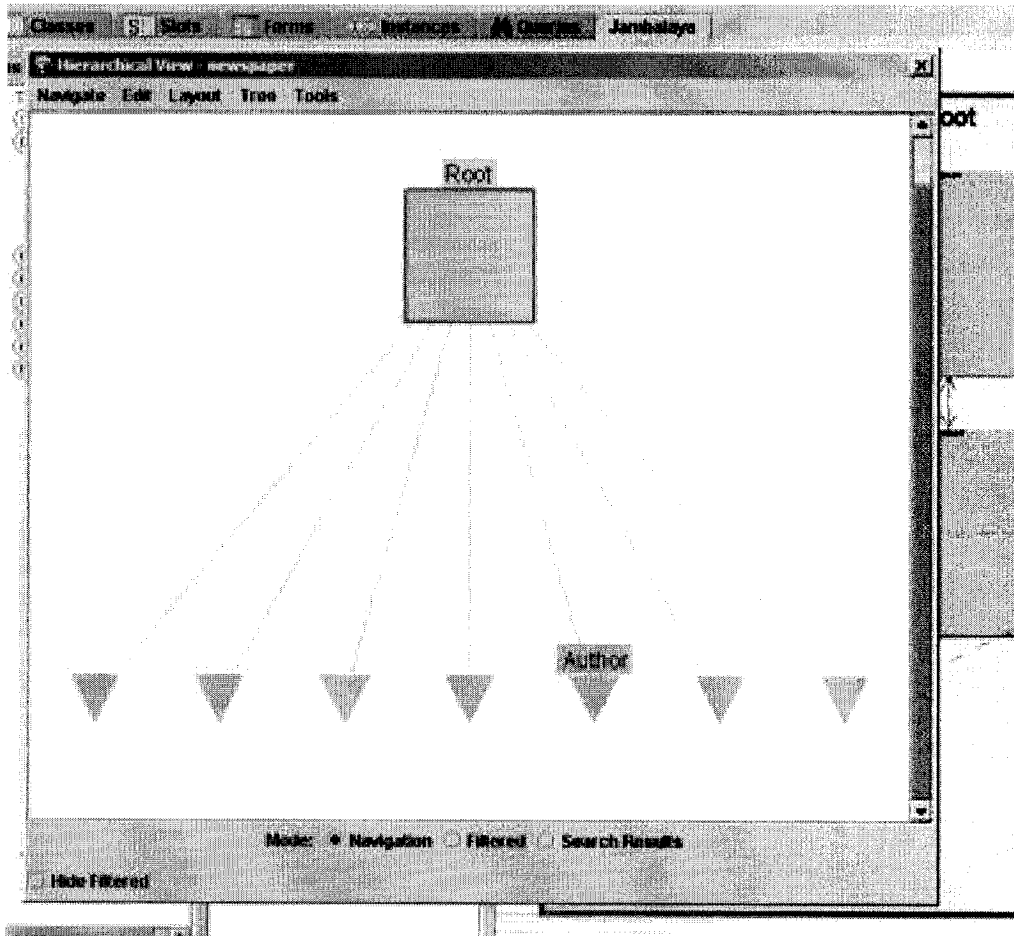

Figure 4: An Example of Hierarchical View in SHriMP

Rigi

Rigi [MULL94] is a system for understanding large information spaces such as software programs, documentation, and the World Wide Web. This is done through a reverse engineering approach that models the system by extracting artifacts from the information space, organizing them into higher level abstractions, and presenting the model graphically. A Rigi view is shown in Figure 5.

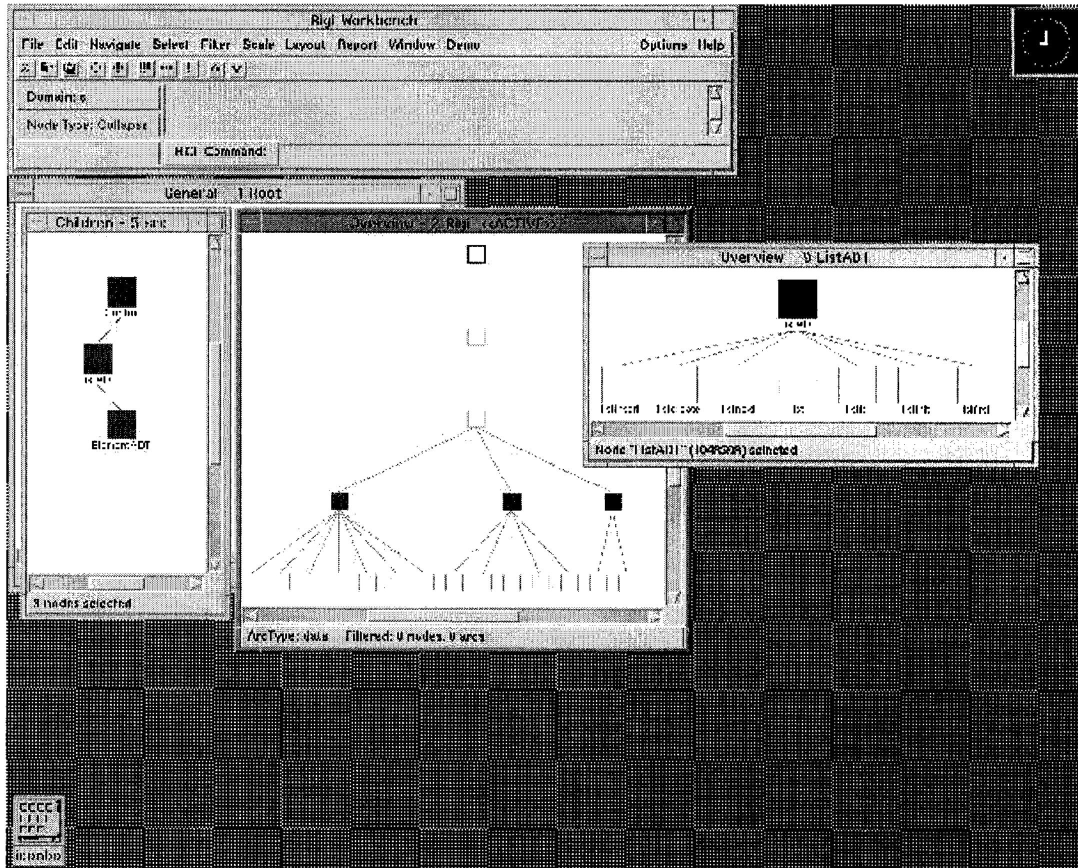

Figure 5: A Rigi view

These tools described above use many 2D visualization techniques such as coloring, highlighting, and layout algorithms etc. Some have started to transform 2D diagram to 3D representations. For example, Imagix4D can generate 3D diagram similar to the 2D ones, using boxes and cones instead of rectangles and lines.

Chapter 3

Software Visualization

Software visualization can be seen as a specialized subset of information visualization. This is because information visualization is the process of creating a graphical representation of abstract. This is exactly what is required when trying to visualize software.

3.1 Visualization

Visualization is an old term which has received a large amount of interest in the Computer Science community. Visualization has previously been defined as the "formation of visual images; the act or process of interpreting in visual terms or of putting into visual form" More recently a new definition has been added: "A tool or method for interpreting image data fed into a computer and for generating images from complex multi-dimensional data sets" [MCCO87]. The main goals of visualization are:

- Maximize human understanding, communication and collaboration without huge effort
- Make evident the meaning of abstract entities
- Reduce the complexity of the phenomena
- Enhance understanding of concepts and processes
- Gain new insight

Visualization is classified as *information visualization*, which hinges on finding a spatial mapping of data that is not inherently spatial, and *scientific visualization*, which uses a spatial layout that is implicit in the data. Both have many applications in areas of automobile industry, flow-dynamics of airplanes, chemical experiments, galaxy simulations, biology and chemistry, and many more.

3.2 Software Visualization

Software visualization is visualization in Computer Science. It has a long history starting with the development of "Flow Charts" in the 60's. In the 70's, "Pretty-Printing" was another major step, followed by "Algorithm Animation" in the 80's. The 90's symbolize the revolution area with the introduction of CASE tools. Finally, in the last several years, the focus is on dynamic data and software processes. Software visualization is currently a hot research area.

There are many definitions of the term software visualization. One that seems to encompass them all is stated by [KNIG99]:

"Software visualization is a discipline that makes use of various forms of imagery to provide insight and understanding and to reduce complexity of the existing software system under consideration"

Software visualization is difficult since software is an abstract element with no spatial representation like a chair or a table. It is somehow hard to project a clear view of what it is and how it actually works in our mind. That's why software visualization is often referred to as the process of "making the invisible visible" [BOEC85]. Gomez

[GOME 01] put it best when he wrote: “In software visualization, there are only two ‘touchable’ realities: the source code and the resulting visualization…the ‘missing-link’ is establishing a ‘cause-effect’ relation between the two entities” [GOME01]. The main challenges of software visualization listed by Young [YOUN96] are:

- Metaphor (representation): how to represent the entities of the software
- Abstraction: what to present and in how much detail
- Navigation: how to move through the sets of visual objects that represent the system
- Correlation: how to link visual abstractions to source code and documentation

Our research focuses on software visualization systems toward supporting large-scale software maintenance. In order to accomplish this task, these systems must focus on the five dimensions of software visualization defined by Maletic et al [MALE02]. These dimensions reflect the why, who, what, where and, how of the software visualization. The dimensions are as follows:

- Tasks – why is the visualization needed?
- Audience – who will use the visualization?
- Target – what is the data source to represent?
- Representation – how to represent it?
- Medium – where to represent the visualization?

These dimensions define a framework capable of accommodating a large spectrum of software visualization systems. Our work on 3D techniques is along the *representation* dimension, while program analysis is along the *target* dimension.

Before moving on to the 3D software visualization, we discuss the limitations of current 2D graph based visualization approaches. In particular, we discuss visualization in the form of reverse engineered 2D diagrams (e.g., collaboration diagrams, call-graphs, sequence diagrams, etc.) and models (UML class models), which have been suggested in the literature [BOOC99] to provide users with higher abstraction views of the software under investigation. The principal aim of visualization is to ease program comprehension by enabling a person to comprehend complex internal structure and entity relationships through appropriate visual mappings. For large software systems, however, it becomes increasingly difficult to comprehend these diagrams for several reasons:

(1) The visualization technique does not scale up causing increased clutter in the diagram because of the large amount of information (entities and entity relationships) to be displayed, essentially resulting in information overload problems. Often, 2D inheritance trees or call graphs of several thousands of entities create space filling and incomprehensible visuals.

(2) The awkward layout techniques provided by the chosen visualization mapping tend to obscure important patterns and relationships in the software from the user.

(3) The navigation tools are non-intuitive; pan-zoom and overlapping multiple windows are typically the kinds of navigation tools supported causing

cognitive discontinuity problems. There are some visualization mappings such as fisheye-views [FURN86], perspective information walls [MACK91], hyperbolic trees [LAMP95] etc., which offer some solutions for focus versus detail. They assist the user in not getting lost in the visual space, but even these do not easily scale to very large software.

(4) Often their scope is rather specialized to depict only certain program artifacts and their relationships.

3.3 3D Software Visualization

3D visualization utilizes advanced visualization and hardware technologies, with the goal to match closely the human cognitive and perceptual views. However, 3D visualization not only introduces an additional dimension, but at the same time also many new challenges. In what follows, we discuss some of the advantages and challenges of 3D visualization.

3.3.1. 3D versus 2D Visualization

As mentioned earlier, over the last decade, programs have become larger and more complex, creating new challenges to the programmer in visualizing these complex and large source code structures. Providing different views might not be sufficient as users are still dealing with a large amount of information and data. Also, not every visualization technique is equally adept in displaying a particular aspect of software structure [NIEL98]. The visualization technique might lack an appropriate navigation support or may not allow the effective reduction of the amount of information displayed through a choice of distinct views. The disadvantages of most

of the commonly used high-level abstractions such as call-graph, UML class models, collaboration diagrams, etc. have been discussed by other authors [RILL01].

Software visualization of source code structures and execution behaviors could consists of both static views and dynamic views [PIRO01, SHNE92, WALK98]. Dynamic views are based on information from the analysis of recorded or monitored program executions. During the recording of a program execution, a large amount of data may be collected. Although this is not a new problem, the rapid increases in the quantity of information available and a growing need for more highly optimized solutions have both added to the pressure to make good and effective use of this information [MALE01].

Three-dimensional visual representations are often suggested and presented as a solution to provide just this required extra space and resulting ease of use in navigation and abstraction level. While the advantages of adding a third dimension are initially obvious, these are realizable only if truly distinct and effective use is made of the added dimension. However, most of the current approaches are just transforming established 2D visualizing techniques into a 3D space [REIS94]. 3D software structure visualizations are still centered on creating standard call graphs within a 3D space. For example, the usage of 3D call-graphs does offer a greater working volume for the graphs thereby increasing the capacity for readability. However, at the same time, they introduce undesirable effects that significantly affect the gain from the added dimension. Problems that might be introduced by 3D visualization techniques include significant objects being obscured, disorientation,

and spatial complexity. To some limited extent, these issues can be resolved by 3D interaction techniques where the viewpoint of the 3D graph is actually within the graph structure; otherwise, the 3D visualization is limited to merely a 2D picture of a 3D structure (See Figure 6) [RILL02].

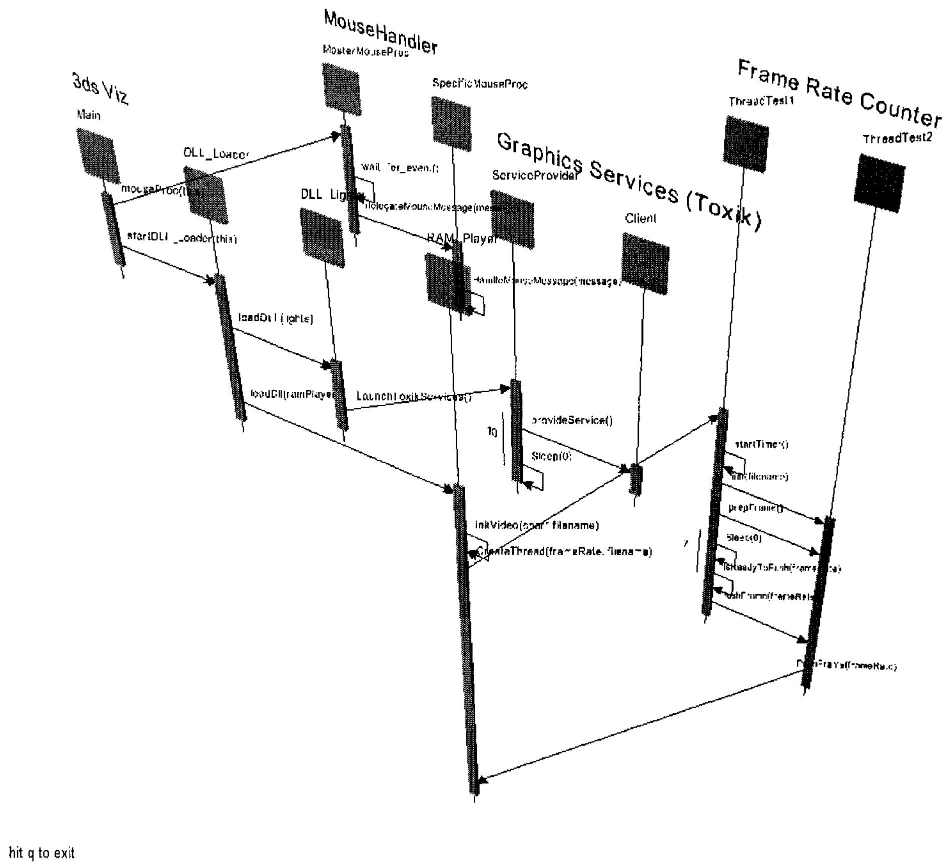

Figure 6: Mapping 2D sequence diagram notation into 3D space

3.3.2. Important Issues

Metaphor

Identifying and selecting an appropriate and intuitive metaphor in visualizing software systems is a major challenge. A metaphor describes a concept in terms of

another concept, where the metaphor should have some similarity or correlation to the concept that will be modeled by the metaphor. With the introduction of 3D software visualization techniques, many new metaphors are being investigated to help in the understanding of large programs. Metaphor creates mapping from software entities to graphic elements. Good and meaningful metaphors ease the complexity of the system and fast the process of building mental model. Shapes, colors, shading, lightning, brightness, shininess and so on can be used to represent some features of the software entities. Positions can be used to indicate the relationships between them. Transparency of objects reveals the notion of "inside".

Mapping program artifacts into the 3D space allows users to identify common shapes or common configurations that may become apparent, and which could then be related directly to design features in the code. Soft City [KNIG00], Cone Tree [ROBE93], hyperbolic tree [LAMP95], and Information Cube [REKI93] are some examples of software visualization techniques that are based on different 3D metaphors. Additional examples for 3D metaphors can be found in [BALL96]. Each of these metaphors has its own pros and cons, and is often only suitable for a specific comprehension purpose. Moreover, a metaphor that is easily understood by one programmer may be confusing to another programmer. Therefore, one of the challenges of current research is to investigate and explore new metaphors.

Orientation and Navigation

One of the most important criteria for a successful 3D system is to provide effective and useful navigation. However, the challenges are to provide an intuitive

navigation without loosing the orientation. In particular one has to address questions, like “Where am I?”, “Where is the target I am looking for?” [HERM00] Changing the user’s viewpoint might provide one possible solution to this problem. The provision of both, a focus and context sensitive view, might help the user to avoid the problem of disorientation and/or to gain re-orientation.

Filtering and Clustering

Although 3D systems can handle in theory a larger amount of information than 2D, they still face challenges in visualizing very large-scale software systems. One has to consider the ability to view, discern information to improve the usability of the representation techniques. To solve the problem, filtering and clustering are used for information hiding and graph simplification. Grouping can be described as a process of program analysis prior to the layout management. The layout algorithms are constrained by the amount of information to be displayed and the limited screen spaces. Even, if one manages to create a layout, the resulting visual might have far too much information, causing an information overload. Therefore, limiting the number of entities to be displayed to the user is one of the key challenges in software visualization. For the visualization of large software systems, it is essential to provide some type of grouping to create a decomposition of the system. It has been shown that grouping can improve readability [MANC99], by supporting a representation that is closely related to the mental model a programmer forms of a system [TZER01] during typical comprehension tasks. Grouping or clustering can be applied to generate

suitable abstraction levels and therefore allow for a reduction of the amount of information to be displayed on the screen.

Automation and Integration

Visualization tools have to address issues related to the automatic extraction information and the resulting visual representations. Furthermore, this will require that visualization tools are tightly integrated with other analysis and extraction tools to be accepted by users. Ideally, visualization tools should be tightly integrated within existing software development a comprehension environments and allow for a seaming less data exchange among the existing tools.

Decomposition and Abstraction

Abstraction of systems should be made at different levels and details. Ideally, the visualization techniques should support the traceability among the low level and higher level abstractions (e.g. between source code and design documents). The visualization approaches also should provide an effective way of switching the perspectives among the different levels of abstractions (views).

User Centered

Over the last several years, user interface design and usability aspects have gained on importance in the software engineering domain. Since the visualization tools are developed to support users in an intuitive fashion during the comprehension of software systems, the usability aspect becomes a major design issue for the adoption of both the visualization techniques and the tools implementing these techniques.

3.3.3. Advantages of 3D Visualization

In what follows, we summarize the major benefits of 3D visualization techniques in comparison to 2D visualization techniques:

- 3D has one extra dimension that can be used to encode some knowledge. 3D graphics have more flexibility to represent and organize the information.
- In 3D environment, users have much greater working volume than in 2D views. Hence it can handle much larger systems.
- Comparing to 2D navigation (on the ground), 3D environments usually provide useful and effective navigation (in the air). It is much like the human's real world experience. Human experience in interacting and exploring within 3D environments is exploited.
- 2D systems usually work in read-only mode. 3D environments provide useful interactions for users.
- 2D representation is usually static while 3D provides not only static information but also dynamic information.
- Overview and context. Being able to display a large amount of information in one view and thus provide an overview.

3.3.4.Challenges of 3D Visualization

"A badly designed three-dimensional visualization is worse than none at all. The extra dimension can open a whole new world of possibilities but at the same time also new challenges."[KNIG00]

While the benefits of adding a third dimension seem to be obvious, these benefits will become distinct only if the visualization techniques explicitly take advantage of the added dimension. However, as we have seen earlier, most of the current approaches simply transform established 2D visualizing techniques into a 3D space [HERM00]. Simply extending "nodes and arc" technique into the 3D space does not necessarily harness the power of 3D software visualization.

Some of the challenges in 3D visualization are:

- **Easier to disorientation.** While 3D system provides convenient navigation and interaction, it also has the problem of disorientation. The user should have a good mental concept of spatial issues like up and down. Spatial awareness is needed for a user to navigate any 3D environment. Notions like "up" and "down" do not conform to our preconceptions of 3D environment and experience and cause confusion. After getting the details of interests, reorientation should be taken into consideration.

- **More complex interface and interaction**. 3D environment has much more complex interface for the user to interact or navigate the system.

- **Layout**. Layout is a headache for visualization. 3D layout algorithm is much more complex and difficult in comparison to 2D layout algorithm. There are many 2D layout techniques like tree map, etc. However, none of them is perfect. Even 2D layout is so difficult and complex, not to mention 3D layout. This is an ongoing research. Progress has been made and we expect to see more and overwhelming improvements in this field.

- **Suitable metaphor**. Software systems are not real world objects. "Software is intangible, having no physical shape or size. After it is written, code disappears into files kept on disks" [KNIG00]. A mapping metaphor creates transformation from intangible software elements to tangible graphic elements. But how to get suitable metaphors is somehow difficult.

- **Information overloading**. It intends to present as much information as possible. It suffers from overloading.

- **More expensive**. 3D views require 3D graphics hardware. Compared to 2D, it is much more expensive because of memory needs etc.

3.3.5. 3D Visualization Techniques

Peter Young [YOUN96] summarized a variety of 3D visualization techniques that are used in many visualization systems. These techniques can be roughly classified three categories: mappings, representations, and dynamic visualization techniques.

- **Mapping from the data domain to the visualization space**

Surface Plots

It is one of the most familiar extensions from standard 2D graphs. Surface plots are constructed by plotting data triples onto the three co-ordinate axes X, Y and Z. Typically the data will consist of two standard sets which have a regular structure, e.g. days of the week and time of the day and one actual data value, for example, wind strength (see Figure 7). The two regular sets are normally plotted on the horizontal axes X and Z with the variable data being plotted as height in the Y axis. The set of points thus formed are netted into a mesh or surface which is often color coded to indicate height variations. The resulting visualization resembles a landscape which can be easily interpreted to identify features such as patterns or irregularities.

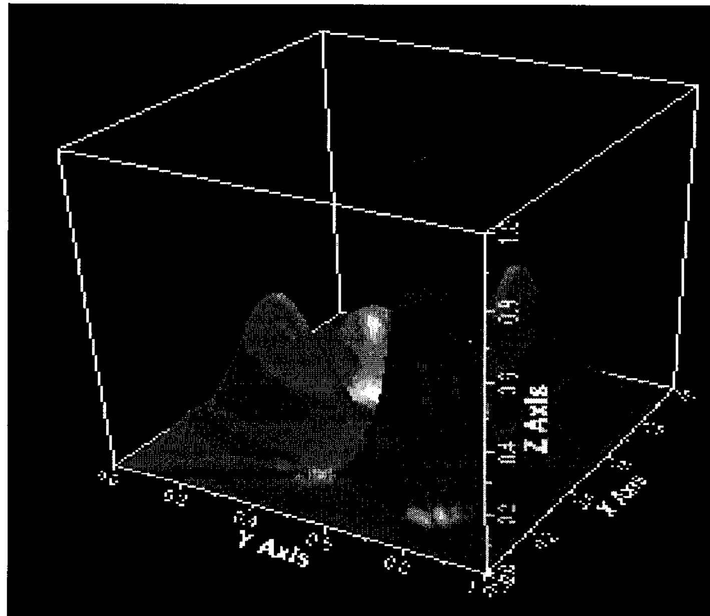

Figure 7: 3D surface plots using Java 3D

Cityscapes

Cityscapes [WALK93] are basically an extension to 3D bar charts and a variation to surface plots. Cityscapes are created in a similar fashion to the surface plots by mapping scalar data values onto the height of 3D vertical bars or blocks, the blocks being placed on a uniform 2D horizontal plane. The resulting visualization is a more granular representation of the surface plot. Figure 8 shows an example of using Cityscape in creating a software landscape [YOUN96].

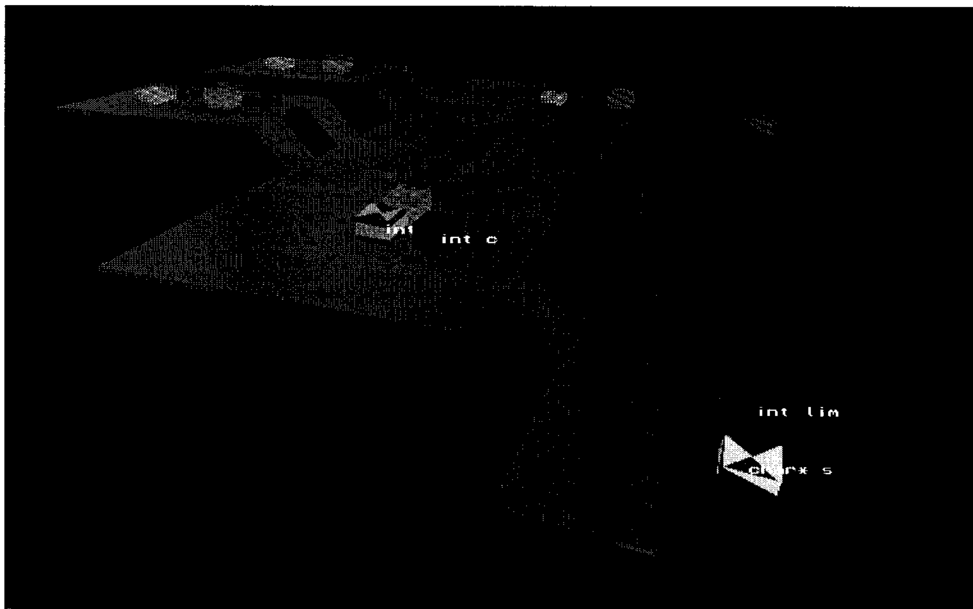

Figure 8: Cityscape for software visualization

Benediktine space

Benediktine space [BENE91] is a term which arose from Michael Benediktine research into the structure of Cyberspace. It maps the objects to

extrinsic dimensions, which specify a point within space and the intrinsic dimensions which specify object attributes such as color, shape, etc. An example of a Benediktine space could be to map an attribute such as student names to the x-axis and their exam marks to the y-axis. The degree which a student received could then be mapped onto an intrinsic dimension such as shape [YOUN96].

Spatial arrangement of data

A key problem is in creating a useful mapping from the data itself to a corresponding representation and location within the virtual environment [BENF94, COLE94]. The requirement of this process is to create a spatial configuration from which the properties of data items within the information terrain and the relationships between them can be readily interpreted simply from their position and presentation.

- **Information representation techniques**

Perspective walls

Perspective walls [MACK91] were used to view and navigate large linearly structured information. This technique employs the space strategy aiming at presenting as much information as possible, and the time strategy that breaks the information structure into several separate views with a switch between the views. Figure 9 shows a class schedule. The front side of the wall shows all the details of the two selected months, while the others are shown on the side walls,

but the user can still get some general information about the months not in front of him.

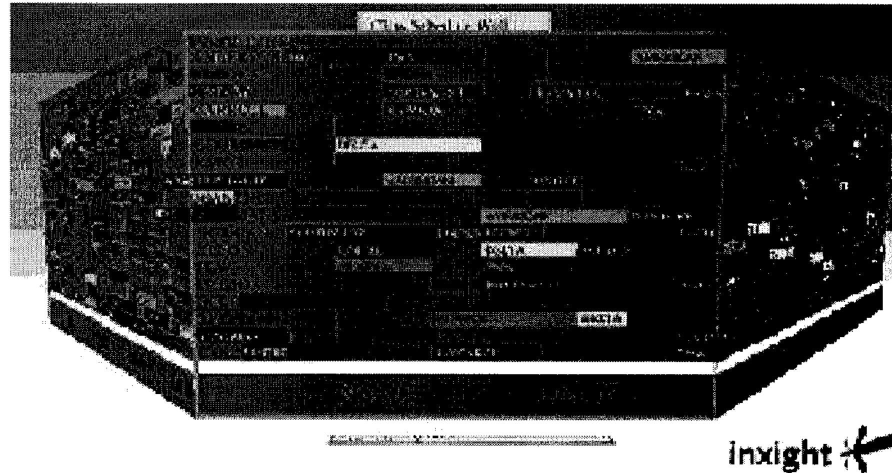

Figure 9: A class schedule on perspective walls

Cone tree and cam tree

Cone tree [ROBE93] is one of the best-known 3D graph layout techniques in visualization. It is a way of displaying hierarchical data (such as org charts or directory structures) in three dimensions. Nodes are placed at the apex of a cone with its children placed evenly along its base. This allows a denser layout than traditional 2-dimensional diagrams. Cam trees are identical to cone trees except they grow horizontally. Cam trees [BALL96] are identical to cone trees except they grow horizontally as opposed to vertically. Figure 10 is a view of a file structure using cam trees [ROBE93].

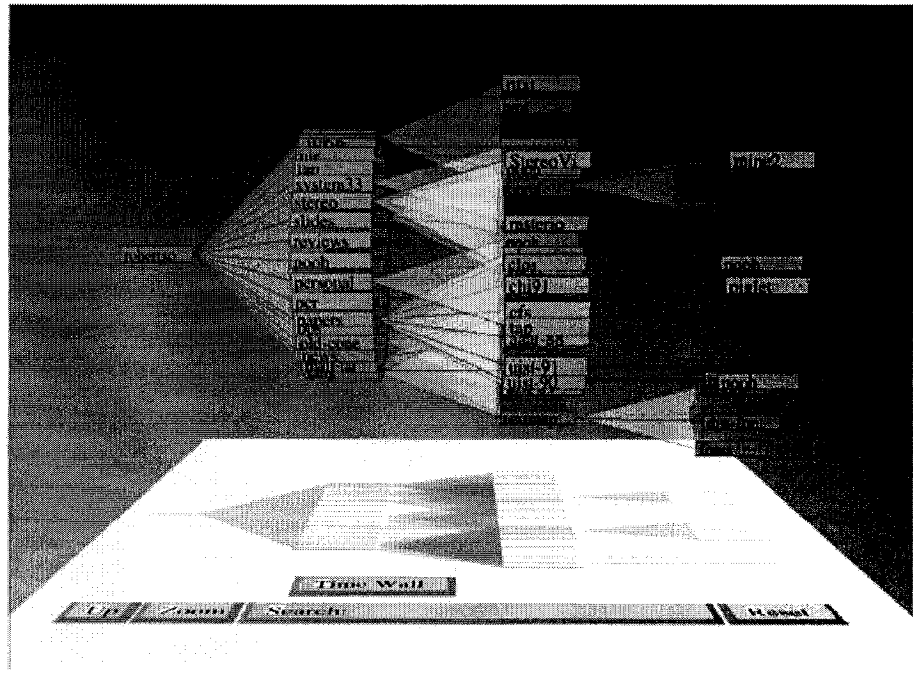

Figure 10: View of a file structure using Cam trees

3D-Rooms

The "3D-Rooms" metaphor is a three dimensional counterpart to the desktop metaphor commonly encountered in computing today. 3D-Rooms are a 3D extension to the original concept of 2D rooms. 2D Rooms [CARD87, HEND86] built upon the notion of a multiple desktop workspace by adding features such as the ability to share the same objects between different workspaces, overview the workspaces and also to load and save workspaces. 3D-Rooms allow the user to structure and organize their work by allocating certain tasks to certain rooms with doors connecting them and floor plans available in another window.

Using this technique, a single result will be displayed as a virtual object in a room. When the result set contains more than one element, several rooms will be used, each with a result in it. These rooms would organize themselves in different structures. The drawback of this approach is that to navigate through the result set, the program needs to choose a property common to each element in this set and use it as the index. This index is then used to arrange the rooms in space and sort the results. This doesn't work very well if the results have only a property. For example if the result set is composed of numbers, it would not mean much to sort these values using themselves as index.

If the properties of the results are more than one, then this kind of visualization becomes useful. For example if the result set is composed of a picture and a number, the latter representing the year in which the picture was drawn, it becomes clear that this approach would build a very nice museum, in which the pictures are ordered by year of their creation.

Information cubes

Information cubes [REKI93] are nested translucent cubes that can be used to denote hierarchical information like packaging. It partitions the available display space into rectangles according to the tree structure. The subdivision represents the relationship of parent and children.

- **Dynamic Information Visualization Techniques**

Fish-eye views

The name given to this particular type of view is taken from the similar effect produced by a very wide angle "fish-eye" lens. Fish-eye views [FURN86, SAKA92] enlarge the focus node with other nodes in lesser detail without losing the whole context when visualizing large graphs. In the limited computer screen, focus + context techniques make it possible to display much information and details which overwhelm the user.

Emotional icons

Emotional icons [WALK95] help to provide a living data environment. It responds to the users' activity, hence making the data world more interactive and dynamic.

Self-organizing graphs

Self organizing graphs typically refer to the techniques used in automatically laying out graphs. Conventional layout techniques involve a function or routine which attempts to perform a suitable layout on a given graph while attempting to satisfy a number of aesthetic criteria or heuristics. Self organizing graphs allow the graph itself to perform the layout by modeling it as an initially unstable physical system and allowing the system to settle into a stable equilibrium according to efficiency, speed, accuracy and aesthetics.

3.3.6. VR and 3D Visualization

Virtual reality (VR) is the simulation of a real or imagined environment that can be visually experienced in 3D and provides a visually interactive experience in full real-time motion with sound, tactile, and so on. VR can produce objects not only from the real world but also objects that do not exist in the real world. The simplest form of virtual reality is a 3D image that can be explored interactively at a personal computer, usually by manipulating keys or the mouse so that the content of the image moves in some direction or zooms in or out. The goal of Virtual Reality is that of creating the illusion of submersion in a computer generated environment. Virtual reality and 3D graphical environment has become commonplace nowadays. Both representations offer the perception of depth. The use of them in computer games is a successful example. The human's perceptual, cognitive and institutive skills can be used in VR. The success of 3D techniques in Virtual Reality gave new life to software visualization.

The distinction between 3D and VR is that a user immersed in a VR environment can always access external information (e.g., the actual source code) without leaving the environment and the context of the representation (e.g., using a palmtop or laptop).

While both representations offer the perception of depth, only VRs allow the user to immerse oneself into the representation. Also, this immersion allows the user to take advantage of their stereoscopic vision. It also helps the viewer to judge relative size of objects and distances between objects. For example, in Figure 11 [MALE01b],

a user immersed in the VR is investigating a visualization of a system. However, VR is not a cure-all. We should not misconceive and exaggerate its power.

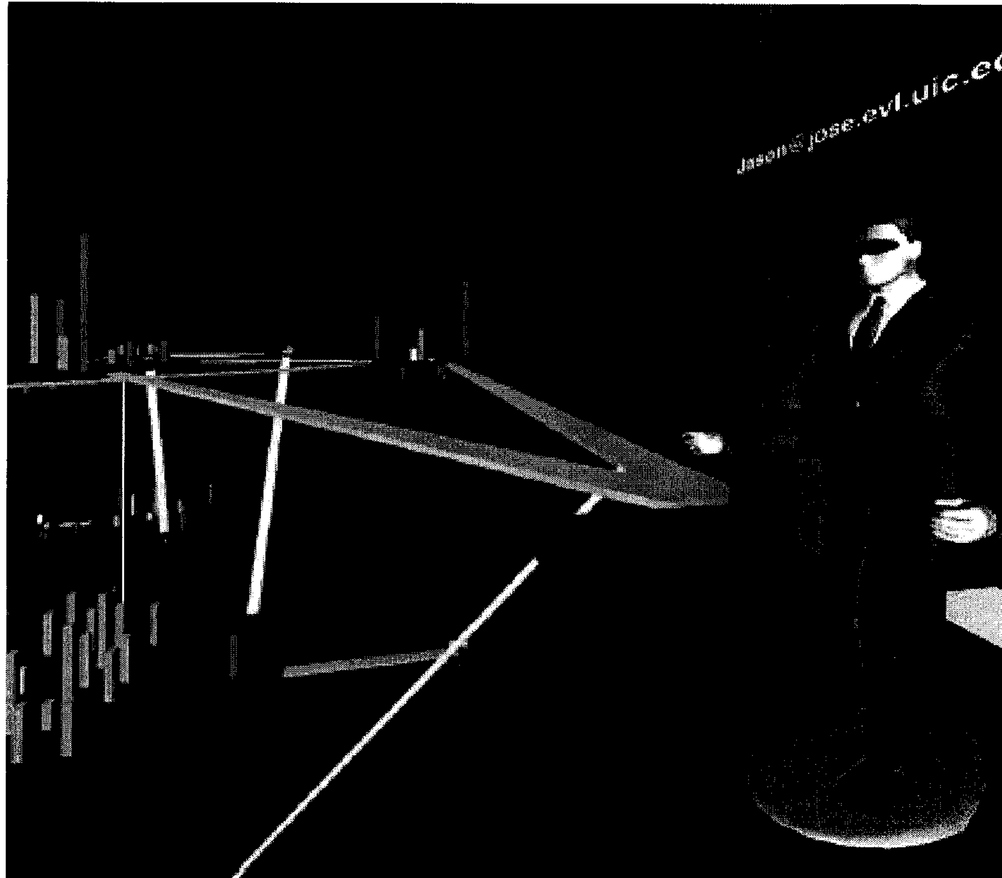

Figure 11: A user immersed in the VR

3.4 Potential 3D Graphic Engines

This thesis also describes a Java implementation of the Virtual City. In order to choose an implementation environment, most importantly, the graphic engine, we first list our considerations: the ability to integrate to CONCEPT, portability, code reuse, performance, the ability of going web browser, coding efforts, and other tradeoffs. And then we examine some potential ready-to-use code libraries, SDKs or APIs, among which

OpenGL, Direct3D, Java3D, and VRML are the most used 3D APIs. We give below a brief description to each of them as well as a comparison among them.

3.4.1.OpenGL

OpenGL, originally designed by Silicon Graphics (SGI) [OPEN 04], now is an open industry standard API for 3D graphics. OpenGL is the successor to the Silicon Graphics IRIS GL library [Sil90] which made SGI workstations so popular. IRIS GL was an API specially designed for SGI workstations. But SGI realized the importance of open standards. Several software and hardware makers took part in specifying an open version of IRIS GL. The Architectural Review Board (ARB) oversees the OpenGL specification, accepts or rejects changes, and proposes conformance tests.

In contrast to the IRIS GL, the OpenGL library is platform and operating system independent. To achieve this independence, all functions for windowing tasks as well as functions for user input were excluded. Special care was taken to easily combine OpenGL with other, platform-dependent, programming libraries, which handle windowing and user input.

Furthermore, the OpenGL library is designed as a streamlined, high performance graphics rendering library. The design is also very hardware near to achieve a reasonable performance. OpenGL is in fact a rendering pipeline, so parts of the pipeline can be implemented either in software or hardware. Many OpenGL primitives can be easily implemented at the hardware level, which of course improves the display speed. Therefore, OpenGL also defines only very primitive geometric objects (points, lines, and polygons). OpenGL library includes functions to specify

most of the scene essentials such a scene camera, geometric primitives, lights, textures, etc.

OpenGL is designed as a state machine, which draws primitives to a frame buffer according to the state of the machine. More than 150 selectable modes can change the state of the machine. Each mode is set independently; setting one does not affect others, although modes interact to determine what is drawn into the frame buffer. Almost all state of the art rendering parameters can be set by different modes. This includes the options for the virtual camera, the light sources, and anti-aliasing. A primitive is a point, line segment, polygon, pixel rectangle, or bitmap, which are the input to the OpenGL machine as shown in Figure 12. Of course there are some functions to perform other OpenGL operations, which do not change the machine state. The output of the OpenGL machine is stored in a frame buffer, which is a memory area to be displayed on the screen. OpenGL supports double buffering, for smooth animations: while OpenGL draws in one frame buffer a second one is displayed, when the drawing process ends the buffers are switched.

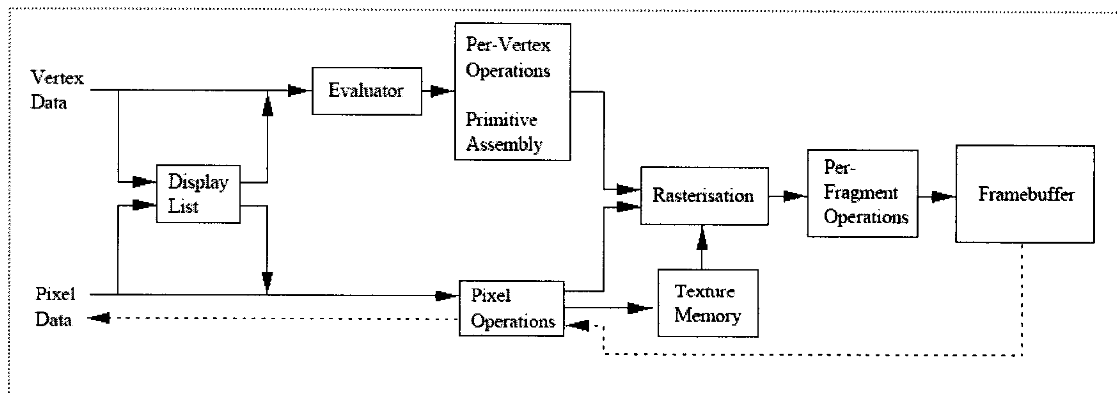

Figure 12: Block diagram of the OpenGL pipeline

There are some OpenGL related APIs which support methods to describe higher-level graphical objects, such as:

GLX & GLU Libraries

The OpenGL Utility Library (GLU) [OPEN 04] contains several routines that use lower-level OpenGL commands to perform tasks such as setting up matrices for specific viewing orientations and projections, performing polygon tessellation, and rendering surfaces. The **OpenGL Extension to the X Window System (GLX)** [OPEN 04] provides a means of creating an OpenGL context and associating it with a draw-able window on a machine that uses the X Window System.

GLUT

The OpenGL Utility Toolkit (GLUT) [GLUT 04], originally written by Mark Kilgard and ported to Win32 by Nate Robins, is a programming interface for writing window system independent OpenGL programs. The toolkit supports the following functionality: multiple windows for OpenGL rendering, callback driven event processing, sophisticated input devices, a "idle" routine and timers, a simple, cascading pop-up menu facility, utility routines to generate various solid and wire frame objects, support for bitmap and stroke fonts, and miscellaneous window management functions, including managing overlays.

Since Java became a popular programming language, bindings between the platform-independent Java programming language and the platform-dependent OpenGL library emerged. An example of such a binding between Java and OpenGL

is JOGL. OpenGL bindings for Java try to provide a complete set of Java bindings to the OpenGL graphics library.

Open Inventor

Open Inventor [OPEN 04] is an object-oriented toolkit built on the top of OpenGL that provides objects and methods for creating interactive three-dimensional graphics applications. Open Inventor provides pre-built objects and a built-in event model for user interaction, high-level application components for creating and editing three-dimensional scenes, and the ability to print objects and exchange data in other graphics formats.

Mesa

Mesa [MESA 04] is a free implementation of the OpenGL API, designed and written by Brian Paul, with contributions from many others. Its performance is competitive, and while it is not officially certified, it is an almost fully compliant OpenGL implementation conforming to the ARB specifications.

3.4.2.Direct3D

Direct3D [DIRE04], Developed by Microsoft, is an API for manipulating and displaying three-dimensional objects. Direct3D provides programmers with a way to develop 3D programs that can utilize whatever graphics acceleration device is installed in the machine. Direct3D Graphics Pipeline is shown in Figure 13. The advantage of Direct3D is that it is faster than OpenGL. Because of this, virtually all 3D accelerator cards for PCs support Direct3D. Thus it is more used in game

development. However, it has drawbacks: (1) Platform dependent (PC Windows platform); (2) Very difficult to use (steep learning curve) with large code overhead compared to OpenGL and Java3D.

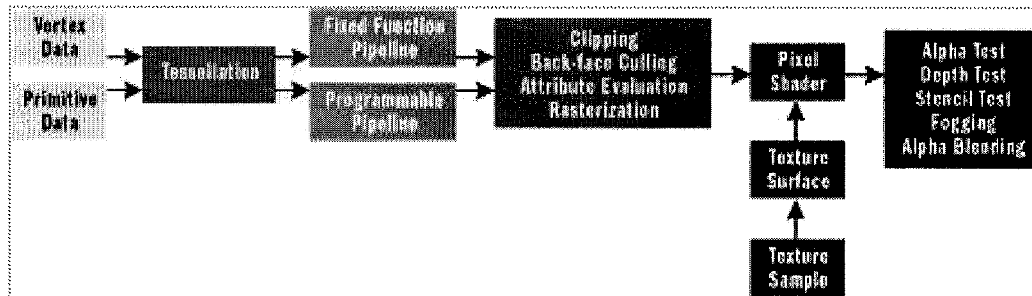

Figure 13: Direct3D Graphics Pipeline

3.4.3. Java3D

As a part of the Java Media product family, the Java3D API [JAVA 04] is an application programming interface used for writing stand-alone three-dimensional graphics applications or Web-based 3D applets. It gives developers high level constructs for creating and manipulating 3D geometry and tools for constructing the structures used in rendering that geometry. With Java 3D API constructs, application developers can describe very large virtual worlds, which, in turn, are efficiently rendered by the Java 3D API. Java3D uses either DirectX or the OpenGL low level API to take advantage of 3D hardware acceleration.

The Java 3D API provides three different modes to render the given virtual universe: immediate mode, retained mode, and compiled-retained mode. The difference between these modes is the possibility to optimize the execution of the

rendering of the actual scene. Each successive mode provides more freedom for optimizations. In more detail, the three modes are:

Immediate mode

Immediate mode provides the lowest level for optimizations. The application does not build a scene graph but provides a Java 3D draw method with all points, lines, and triangles which should be displayed in the selected scene. These graphical objects are then rendered by the Java 3D render.

Retained Mode

Within retained mode the application has to build a scene graph which describes the complete scene. To optimize the execution of the rendering the application has to specify which elements of the scene may change during runtime.

Compiled-Retained Mode

Compiled-retained mode provides the highest level for optimizations at the scene graph level. Like retained mode, the application has to build a scene graph and has to specify which elements of the scene may change. To optimize the execution of the rendering process some parts of the scene graph can be compiled into a Java 3D internal format.

It is recommended to use the retained and compiled retained modes to take advantage of the convenience and the performance benefits these modes provide. A typical Java3D scene graph structure is shown in Figure 14.

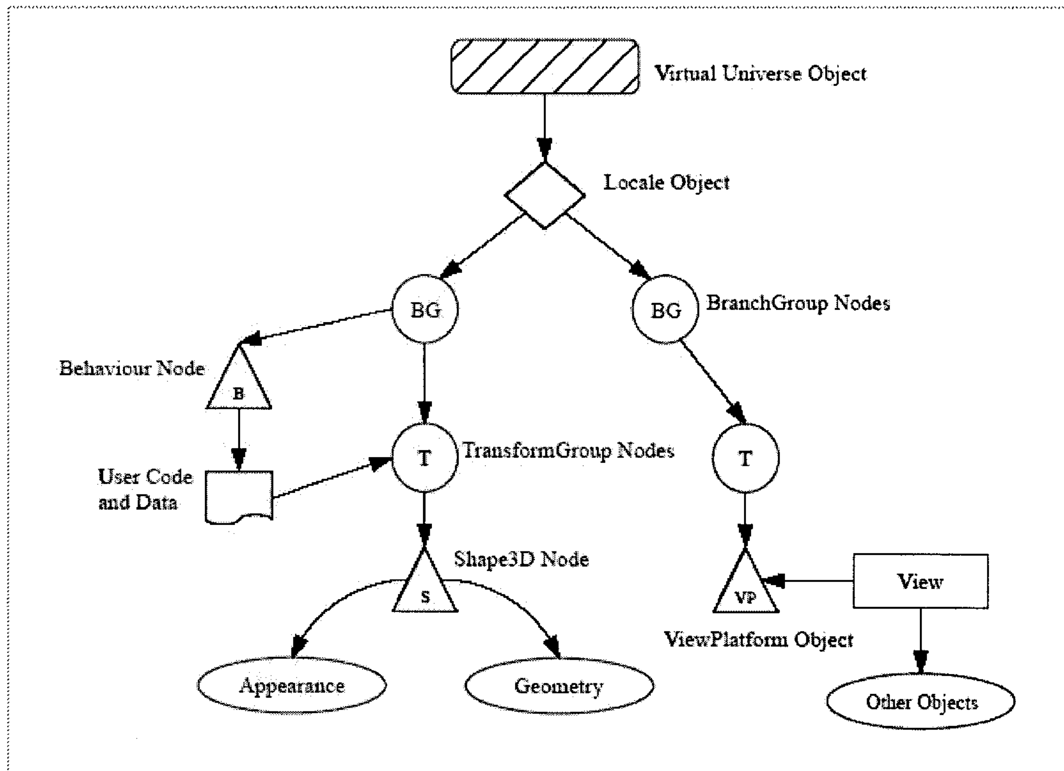

Figure 14: Java 3D application scene graph.

A scene graph is a simple and flexible way to represent and render complex 3D environments. It contains a complete description of the virtual 3D universe. As can be seen in Figure 14, the scene graph is a directed acyclic graph, whose root node is a *VirtualUniverse* object. It is possible to create multiple universes, but most applications will use only one. This node provides the base of the scene graph. Every scene graph must be connected to a *VirtualUniverse.* The children of a *VirtualUniverse* are *Locale* objects. These objects define the origin of the attached sub graphs in high resolution coordinates. A *VirtualUniverse* can hold as many *Locale* objects as needed. The scene graphs itself start with *Group* nodes. *Group* nodes have a variable number of child nodes including other *Group* nodes, but

exactly one parent node. Many operations on *Group* nodes are possible, such as adding, removing and enumerating the children of the node. Many different *Group* nodes for special purposes are available, like the *BranchGroup* node. The *BranchGroup* node is the root of a sub-graph of a scene, attached to a *Locale* or another *Group* node. Another *Group* node is the *LOD* node, which contains an ordered list of children and a level-of-detail value for every child. At the end of every branch there are *Leaf* nodes attached. These nodes are simply nodes which have no children. *Leaf* nodes specify things like lights, geometry, sound, and a view platform for positioning and orienting a view in the virtual world. Very important *Leaf* nodes include the *Behavior* nodes, which provide the means for animating objects, processing keyboard and mouse inputs, reacting to movement, and enabling and processing pick events. These nodes contain Java code to interact with Java objects; values within a Java3D scene graph can be changed, and other computations performed.

Despite its major disadvantage of being too slow for PC-resident simulation application (Java Byte code is interpreted by the Java Virtual Machine, which then talks to Lower Level OpenGL Layer), Java3D has many advantages:

- Platform Independent
- Ability to distribute over the web (3D Applets)
- Small Code size (similar to OpenGL overhead)
- Also-Magician is a front-end to OpenGL
- Java Language: NO Pointers (No Headaches)

3.4.4. VRML

The Virtual Reality Modeling Language (VRML) [VRML04] is a data format to describe interactive, three-dimensional objects and scenes which are interconnected via the World Wide Web. Figure 15 shows a short code example which creates a box. It defines a *Shape* node for the box in a *Transform* node to locate it.

```
#VRML V2.0 utf8

#*** WORLD ************

Viewpoint {
  position 50 10 150
}
Transform {
  translation    50 0 50
  children [
         Shape {
                appearance Appearance {
                       material Material {
                              diffuseColor .1 .2 .3}}
                geometry Box {
                       size 20 40 20
                }
         }
}
```

Figure 15: VRML code for creating a Box

VRML is capable of representing static and animated objects and it can have hyperlinks to other media such as sound, video, and image. Interpreters (browsers) for VRML are widely available for many different platforms as well as authoring tools for creating VRML files. The VRML worlds can also be viewed through a conventional web browser with the aid of a plug-in, for instance, Cortona VRML Client [CORT04]. These VRML browsers have various navigational tools to let the user interact with the virtual environment. For example, Figure 16 shows a virtual city environment, while Figure 17 is a model of a real university campus.

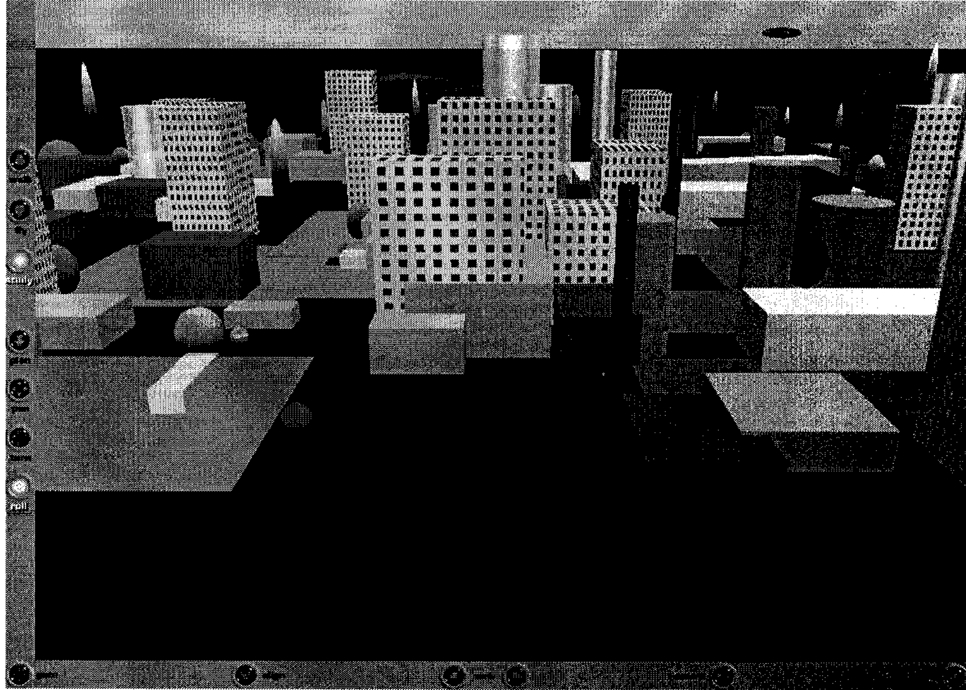

Figure 16: A VRML Virtual City in Cortona Browser

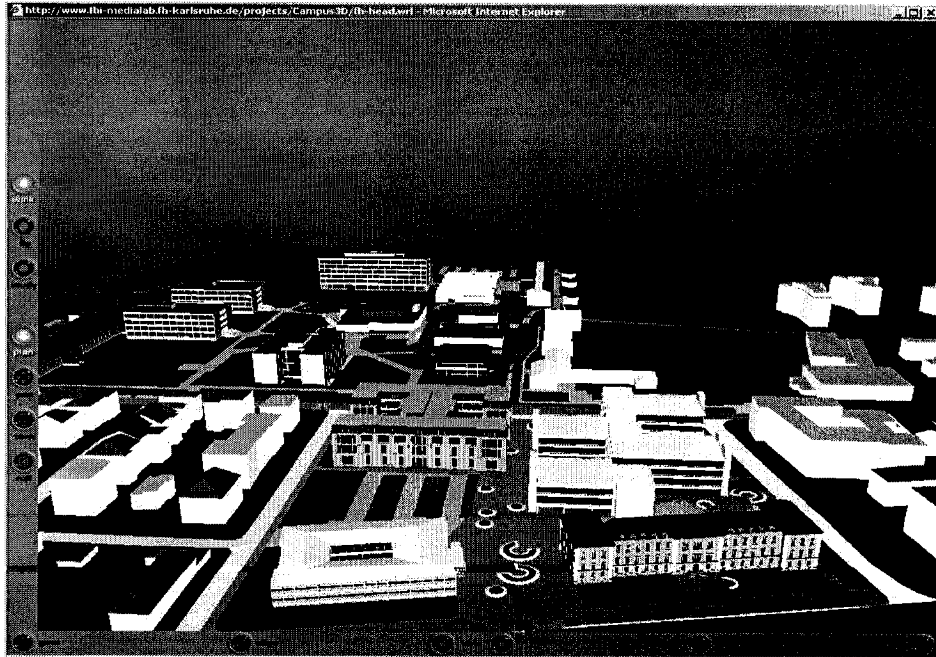

Figure 17: A VRML Campus Browsed with Cortona VRML Client

VRML supports an extensibility model that allows new objects to be defined and a registration process to allow application communities to develop interoperable extensions to the base standard. The most exciting feature of VRML is that it allows the creation of dynamic worlds and sensory-rich virtual environments. This enables its potential abilities of visualizing the history of software.

3.5 XML in the Visualization Pipeline

Extensible Markup Language (XML), a simple and flexible text format, is playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere. The visualization pipeline is a convenient metaphor for the representation, transformation, and presentation of data. The pipeline is constructed by connecting data objects and visual objects.

There are various reasons to the use of XML. One of the most obvious reasons is that XML tells what kind the data is, not how to display it. Because the tags identify the information and break up the data into parts, a variety of applications, services, and so forth, can process it. Because the different parts of the information have been identified, they can be used in different ways by different services as necessary. It is also a benefit when it comes to generating information that a service then makes available to other services.

An additional benefit is that XSLT can be used to automate to some degree the transformation of the raw XML into only those tags that it wishes to deal with. Another strong advantage of using XML rather than byte streams or text files is that the content can be validated by using DTDs.

In one word, the main benefit of working with XML is that it is emerging as the standard for information and data exchange between (distributed) application and services. Through the utilization of XML, the transfer of data to the visualization then just becomes another of the data consumers in the wider system. The XML pipeline makes the visualization seamlessly integrate with other parts of the CONCEPT project.

3.6 Selecting the Appropriate Graphic Engine

The Java languages are chosen for its cross-platform appeal and the possibility to convert the application into a Java applet running in a web browser in the future. After comparing the 3D APIs (see Table 3), the Java3D API is used because it is platform independent and supports a wide range of geometry file formats. The VRML language provides an 'External Authoring Interface' (EAI) that can be used to control complex behavior in the VRML world through JavaScript. This feature enables the application potential future extension to World Wide Web.

Table 3: Comparison of 3D APIs

	OpenGL	Direct3D	Java3D
1. PC Driver Support for graphic accelerator cards	Poor	Good	Very Poor. It is designed to run on top of the other APIs, so should be a superset of them when all are supported in a PC environment
2. Speed	Excellent	Good	Poor. Interpreted
3.Ease of Coding	Good	Poor	Fair
4. Multi-Platform support	Good	Very Poor (PC only)	Excellent
5. Java support	Poor	Very Poor	Excellent
6. Documentation	Fair -	Fair	Fair
7. Ease of Drawing Text	Poor in base standard - you have to use bitmaps. Good if you use GLUT.	Good	Fair
8. Ease of Running in a Window	Poor	Excellent	Not really applicable

3.7 Related Work in 3D Software Visualization

3.7.1. Software World

The Software World [KNIG00] is a visualization targeted at Java code that encompasses real world items based on city and urban development and cartography in an attempt to deal with visualizing software systems of different sizes in a coherent manner. It maps the "Software World" as the whole software system where countries stand for the packages in Java, a city represents a file from the software system, a district symbolizes a class, a building portrays a method, and a monument depicts a class variable. The streets, which connect the districts, illustrate the relationships between classes. In Figure 18, a district shows many small methods and three larger ones represented by buildings.

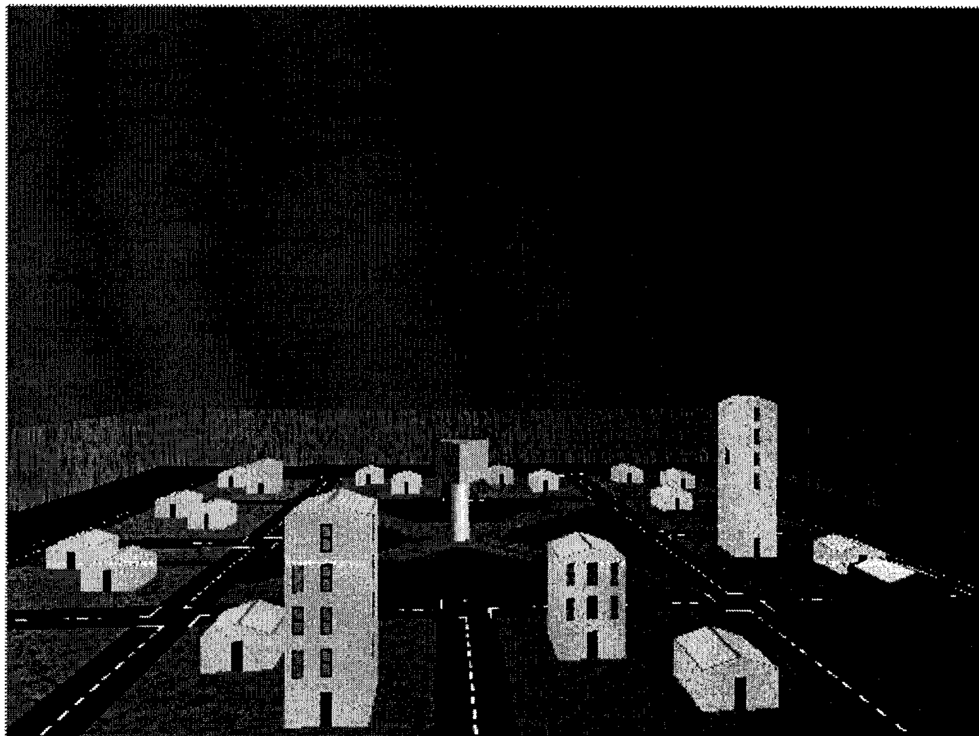

Figure 18: An overview of a district in Software World

It has a prototype implemented using Java and MAVERIK [MAVE 04]. MAVERIK is a publicly available virtual reality (VR) system and has been under development by the Advanced Interfaces Group since 1995. More information about MAVERIK can be found in [MAVE 04].

3.7.2. sv3D

Source Viewer 3D (sv3D) [MALE03] is a software visualization framework that builds on the SeeSoft [EICK92] metaphor. It brings a number of enhancements and extensions over SeeSoft-type representations. In particular it creates 3D renderings of the raw data and various artifacts of the software system and their attributes can be mapped to the 3D metaphors at different abstraction levels. It implements improved, object-based user interactions, while it is independent of the analysis tool accepting a simple and flexible input in XML format. The output of numerous analysis tools can be easily translated to sv3D input format. The sv3D is implemented in C++ and uses Qt [TROL04] for the user interfaces and Open Inventor [OPEN04] for 3D rendering.

Figure 19 shows a 3D overview of a small system with 30 C++ source code files and approximately 4000 lines of code using sv3D. Each file is mapped to one container. Each container is made up of a number of poly cylinders. Each poly cylinder represents a line of source code.

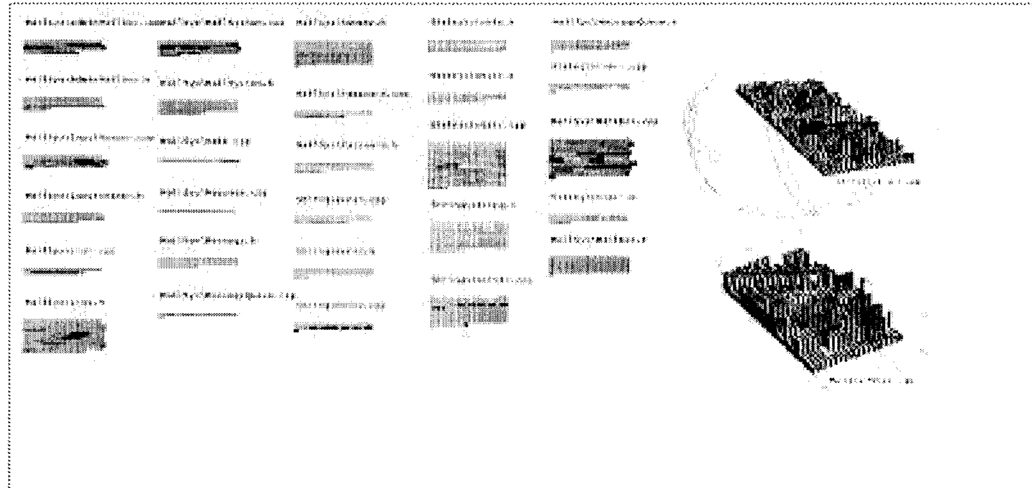

Figure 19: Overview of a system in sv3D

While Figure 20 shows only one file. The container represents the file; each cylinder represents a function; the color represents the hit count for each function; the height of the cylinder represents execution time of the function. A handle box manipulator is active on the container.

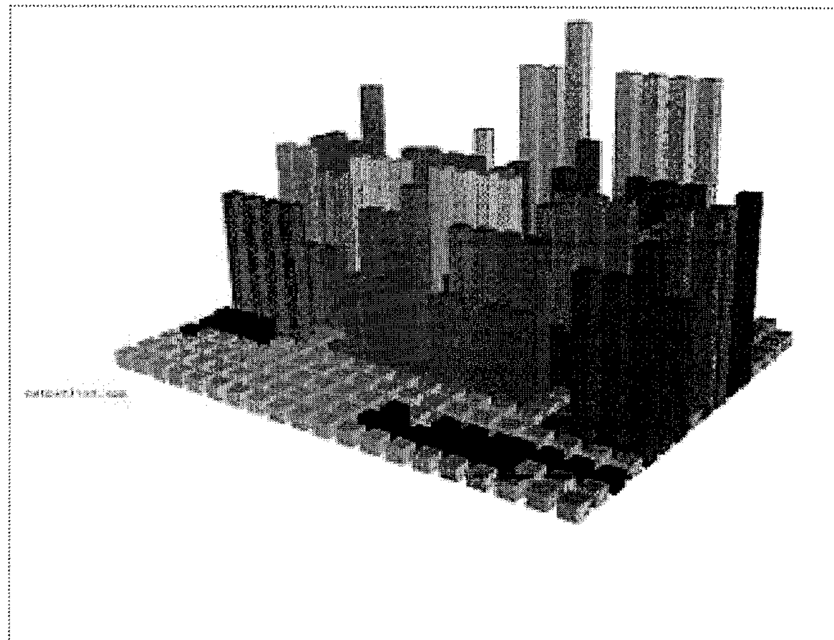

Figure 20: View of one file in sv3D

3.7.3. VizzAnalyzer

The VizzAnalyzer [WELF03] is a framework allowing the combined use of analysis methods and metaphors to efficiently identify architectural entities. The advantage is the decoupling of analysis and visualization, supporting specific tools and processes for each. Analysis and visualization techniques fit well together and accomplish each other through well-defined interfaces. These interfaces allow the individual development, and enhancement, of visualization and analysis techniques, in the way that they can be just exchanged or plugged into the framework.

Some parts of the implementation of the VizzAnalyzer prototype use Java3D as the graphic engine. It maps the graph representations of the low level and high level analysis results to a Java3D *SceneGraph*. The example in Figure 21 shows a view of a package hierarchy using cube and sphere metaphor.

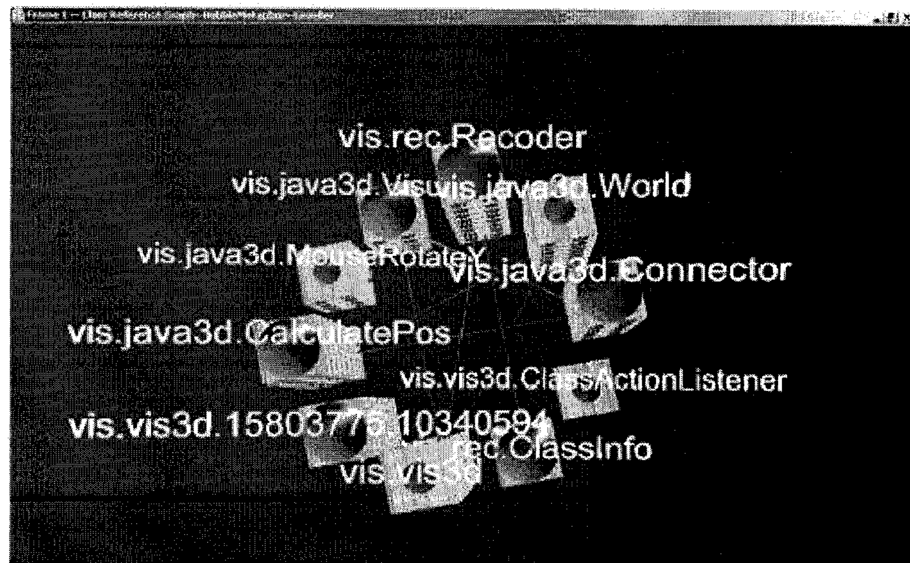

Figure 21: A package hierarchy using the fully qualified class names.

The VizzAnalyzer tries to provide the user with some choices of different metaphors and layout algorithms [PANA03], whereas the 3D City metaphor can be one of the choices. For example, In [PANA03], Thomas Panas et al. suggest that Figure 22 illustrates both static and dynamic information about a program. From a static point of view, the size of the buildings gives the user an idea about the amount of lines of code of the different components. The density of buildings in a certain area shows the amount of coupling between components, where the information is retrieved from metric analysis. The quality of the systems implementation within the various components is visualized through the buildings structures, i.e. old and collapsed buildings indicate source code that needs to be refactored. However, this 3D city implementation is based on 3D Studio Max [DISC04], which is a widely used 3D tool by professional artists and designers to create visual effects, games, and designs. So it is still far away from automatically generating a 3D city scene.

Figure 22 Top view of a 3D city with business information

3.7.4. MetaViz

MetaViz (Metaball Visualization) [WANG03], a plug-in of the CONCEPT project [RILL02], is a software visualization system that uses Metaballs. Meatball is a 3D modeling technique, which has already found extensive use representing complex organic shapes and structural relationships in biology and chemistry, to provide suitable 3D visual representations for software systems. CBO (Coupling Between Objects) is the measurement of internal relationships among software artifacts. It is used to measure design and code quality by investigating the coupling among classes. MetaViz is useful for showing the couplings of software artifacts. Figure 23 shows the coupling relations in a system.

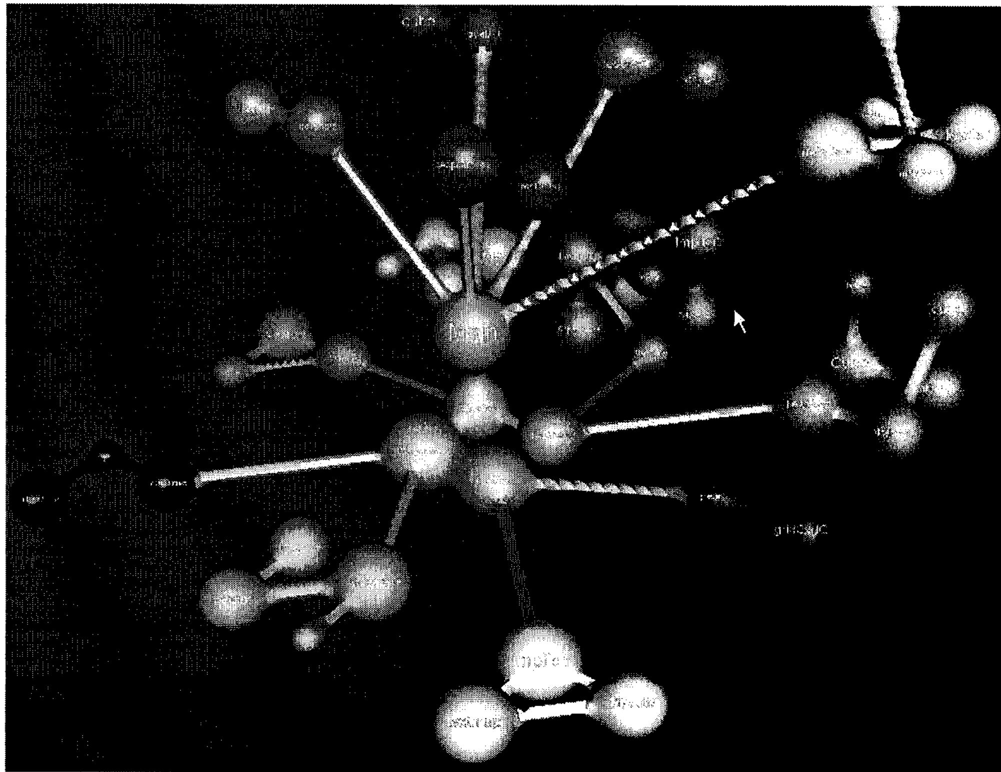

Figure 23: A view of CBO in MetaViz

3.7.5. UML3D

UML3D [XIAN03], also included in the CONCEPT project [RILL02], applies 3D visualization techniques to UML by taking the advantages of 3D space. It integrates a self-organizing layout algorithm for both traditional 2D UML and 3D UML diagrams. The use of layout algorithms can reduce the complexity of a graph and facilitate the task of program comprehension. Moreover, UML3D addresses some other shortcomings of UML by providing intuitive navigation and interactions with the diagrams. Figure 24 shows a system with over 200 classes.

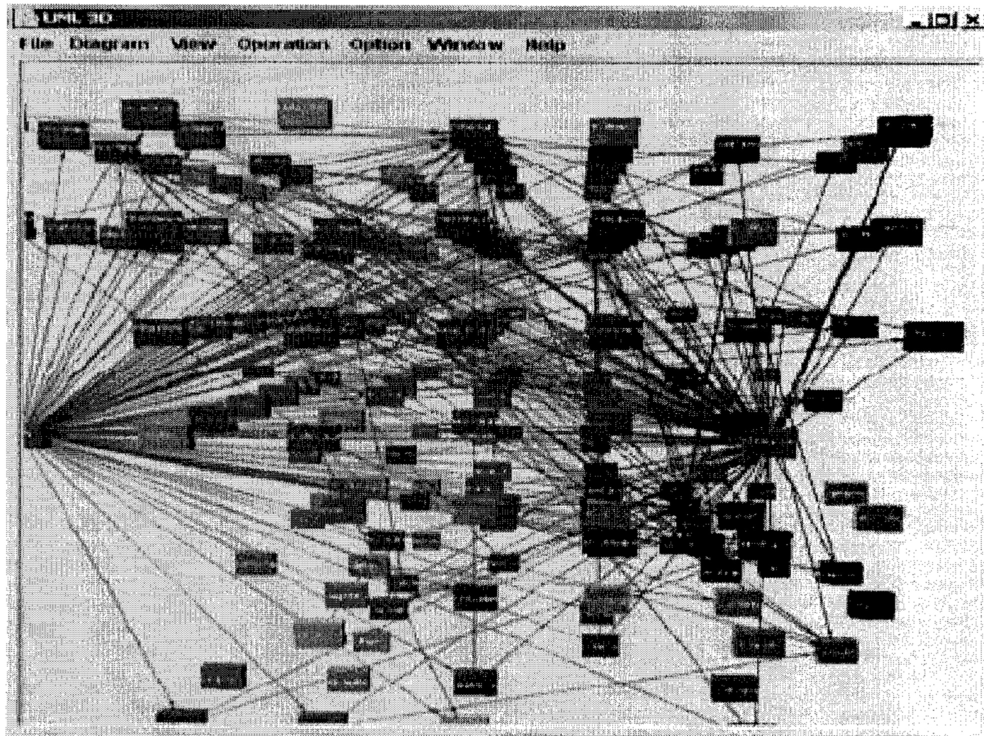

Figure 24: A class diagram of a system with over 200 classes

In summary, these related researches illustrated different approaches of using three dimensional visualization techniques. Software World is the first implementation

using the real world metaphor. The sv3D is highlighted by flexibly displaying different levels of abstraction. VizzAnalyzer is characterized of the claim to combine program analysis and visualization together. Both MetaViz and UML3D show the ability of automatically generating appropriate pictures, while MetaViz examined a novel metaphor.

Chapter 4

The Java3D Virtual City

This chapter introduces first some of the reasoning in selecting the Virtual City metaphor for the visualization of the software structures. We then describe the JAVA 3D implementation of the Virtual City referred to as JVC, and its integration within the CONCEPT environment.

4.1 The Metaphor: the Virtual City

4.1.1.Why Virtual City?

Metaphors, when depicting real world and establishing social interaction [RUSS00], especially in virtual reality, become very important in developing successful visualization tools. The major challenge lies in selecting a metaphor that will improve the usability of the visualizations created. One fundamental problem with many visual representations is that they have no intuitive interpretation, and the user must be trained in order to understand them. Metaphors found in nature or in the real world [KNIG00] avoid this by providing a representation that the user is already familiar with.

The selection and use of metaphors [KNIG00, RILL02, WELF03, PANA03, XIAN03] is not without controversy, because identifying the suitability of a metaphor is a very subjective matter [KNIG00b]. For this research we selected the Virtual City

metaphor because it is based on a metaphor (city representation) most users are familiar with from their daily life. Further, this metaphor provides some new approaches to address the challenges of software visualization such as scalability, navigation, visual complexity and so on.

4.1.2. The Virtual World Metaphor

In what follows, we first introduce and discuss some cognitive aspects related to the structural understanding, using a city landscape metaphor.

In his book "The image of the city" [LYNC60], Kevin Lynch describes a study that looked at how people build a mental representation of a city they live in. Lynch isolates five distinct elements of the mental representation of the city, which he calls the "environmental image". The five elements of the environmental image are *paths, edges, districts, nodes, and landmarks*. More detail is given as follows.

A **"node"** is a distinct location in the environment. This may be a place, a piece of a forest, a room or whatever - it is a strategic spot in an environment the user can enter.

A **"path"** is a channel along which the observer moves. For many people paths are the dominant environmental elements as people often remember spatial concepts in terms of paths. Paths connect nodes or lead to nodes.

"Edges" are borders perceived in the environment. The perception of a linear element as border or path depends mainly on the perspective of the observer - what is a path for the one person may well be seen as edge by another person. For instance

what is seen as path by a car driver is seen as edge by a pedestrian trying to cross the street.

"Districts" are sections of cities perceived as one area because the objects (buildings) show common character. That common character can be functionality provided in the area, like in a harbor area, or the age of houses, like in the center of old cities. Districts often are surrounded by paths or edges.

"Landmarks" are strong points of reference often to be seen from far away.

The discussion above reflects the psychological perspective of how people learn and comprehend a city image. Learning a city image is a long and complicated process. The design of a city has a significant influence on the travelers' ability to navigate and comprehend the layout and representation of the city. Is there a certain set of guidelines that can provide guidance on how paths shall be designed and cities laid out to give the observer a sense of the whole structure - to help building a visual abstraction of the city?

Indeed there exists such a set of "guidelines". The most important rule is to heighten the visual identity of city elements and to structure the environment in a clear way. Paths among buildings have to act as key lines that should show a singular quality. A visual hierarchy of paths and edges has to be provided. Paths and edges should exhibit a sense of directionality. Lynch summarizes these and other rules in a set of 10 guidelines [LYNC60]:

1. "Singularity or figure-background clarity and sharpness of boundaries". This guideline aims at the qualities of city elements which make them identifiable.

2. "Form simplicity". City elements should strive for clarity and simplicity of visible form in the geometrical sense as these forms are more easily incorporated into city image.

3. "Continuity" in edges or surfaces, nearness of parts, repetition of rhythmic intervals.

4. "Dominance of one element in an ensemble" translates to a cluster of elements grouped around one major element. Lynch calls this process "abstraction".

5. "Clarity of joint" means high visibility of joints and seams.

6. "Directional differentiation" means that city elements should exhibit directional qualities. These qualities differentiate one end from the other and are very useful in structuring on a larger scale.

7. "Visual scope" is quality of the environment which influence the range of vision - be it actually or symbolically. Examples are transparencies, overlaps, vistas, panoramas and several others.

8. "Motion awareness" argues that actual and potential motion shall be made explicit to the observer.

9. "Time series", for instance sequences of landmarks, are series of elements which are sensed over time.

10. "Names and meanings" are non-physical characteristics which may enhance the elements more imaginable. They sometimes give clues about the location (like in "North Station") or trigger historical, functional or economical associations.

Adherence to only one or two of these guidelines will not make an environment easy to learn - instead it is the total orchestration of these units which can knit together a dense and vivid image and can help maintain this image over areas even of metropolitan scale. If environments feature a strong visible framework and highly characteristic parts the exploration of new sectors is easier and more inviting. Going back to the already known areas then will communicate a feeling of "home".

The environment acts as the information carrier. Objects in space give information about their usage. Many objects are formed in a way that it is evident what they are meant for. In order to establish a meaningful mapping, we have to examine the real word properties and structures that can be modeled.

From the view point we adopted in this research, the world can be represented at different levels of abstraction, which can be briefly described as

World, flattened, overview picture, atlas style, not showing necessarily all the countries as it would be known in standard geography. The world view would show

however the major participants (continents) at a very high level and the relationships between those elements.

Continent, a continent is a group of countries. At this level of abstraction, the user will be able to see all the countries associated with the particular continent.

Country, Each continent consists of one or many countries. The view provides a way of splitting the items in the world down one level without the detail that is provided by the next level down.

City, are the next level of detail. Each country consist of or more cities. These cities are composed of sub-areas but to ease the navigation and complexity of the visual, only the larger districts might be shown in this view.

Districts, there can be several of these in a city, the number depends on the level of information detail to be presented. Districts group together related aspects of the city and provide groupings to be used when moving from a higher level of abstraction to a more detailed level.

Streets/Buildings/Gardens/Monuments, these show the detail of the visualization and provide the next level of abstraction down from the districts. They also act as legibility features and landmarks of the city.

Inside Buildings/Gardens, this is the finest level of detail, where detailed direct mappings from the code to the visualization can be made.

4.1.3.The OO Program Artifacts

An Object-Oriented program is a collection of ***Classes*** and ***Objects*** of those classes. Objects are instantiated from classes during the execution of a program. So they are components of a dynamic system. In a program, there will have variant numbers of classes. Four types of relationships between classes are:

Association: it is a bi-directional connection between classes.

Aggregation: it is a relationship between a whole and its parts.

Dependency: it is a relationship between a client and a supplier where the client does not have semantic knowledge of the supplier.

Inheritance: it is a relationship between a super class and its subclass.

These visualizations can theoretically be used to visualize any modern object-oriented although in practice this does not work due to differences in the structure of languages such as C and C++. Our work currently focuses on the visualization of Java code. The items of interest in this language are listed in Table 4.

Table 4 Language issues of OO programs

Item	Properties
Files	Name Size Location
Packages	Name Files
Classes	Name Extends from and implements (inheritance information) Package contained in File contained in Accessibility (which modifiers are used) Imported packages Is inner class **Properties can be expressed numerically**: Number of constructors Number of methods Number of attributes Number of children Number of parents Lines of code Age of code
Methods	Name Parameter names and types Return type Exceptions thrown Usage Accessibility (which modifiers are used) Number of arguments Number of lines of code Is Static
Attributes of a class	Type Value at declaration (if any) Accessibility / Protection Is Static Is Array Usage (?) Is constant
Method / Function (local) variables	Name Type Value at declaration (if any) Usage Accessibility (which modifiers are used) Is array Is constant Scope within the method/function

These items form the basis of the artifacts that will be visualized. There is also a hierarchy implicitly imposed on the information because of the design of the Java language. Each piece of code must be in a package and a class, so complete ordering information exists. Packages contain classes and classes contain methods, attributes, etc.

All of the items of interest can be obtained from a static analysis of the source ode. Further analysis on the result can abstract higher level information. The aim is to set a coherent visualization system to visualize these items and higher level abstractions.

4.1.4. Mapping the Virtual City Metaphor to OO Artifacts

The OO languages elements can be mapped to different visualization levels as shown in Table 5.

Table 5 Mappings

Visual objects	Program artifacts	Description
World	A system as whole	Shows the different systems the current system is interacting with.
Continent	Group of countries	A system and its major subsystems (overall architectural view)
Country	Group of cities	A subsystem view
City	A subsystem	A particular program/subsystem
Block / District	A group of classes	Classes with **Inherence** relationship can be grouped into a block or adjacent blocks
Building	Class.	Each building has orientation.
Floors	Methods (and constructors).	Use colors to identify **Private, Public** Use transparence to indicate **Virtual**
Entrance	Return type and name	Return type and method name can be shown on the entrance door.
Windows	Arguments list	Windows spread on walls represent parameters. The number of windows directly correlates the number of the arguments.
Path/Road	Relationship between classes	Association etc
Height of a floor	Number of lines of code in a method	

4.2 System Overview

In this thesis we combine 3D visualization and navigation techniques as suggested in [BALL96] with different types of program analysis [RILL02b]. The combination of program analysis [RILL02b, WELF02] and software visualization techniques is crucial to succeed in comprehending larger systems and their architectures. Program analysis not only provides additional insights, interpretation, and filtering techniques for the

information to be displayed, but also provides additional challenges with respect to visualization and graphical represent this additional information in an intuitive form.

4.2.1. Proposed Approach

We propose the use of 3D visualization techniques and program analysis techniques to enhance the expressiveness of the software visualization representation to guide programmers during the comprehension process of larger systems.

Software visualization has been widely used by the reverse engineering research community during the past two decades [STOR97b, STOR98, STOR01]. Many of them provide ways to uncover and navigate information about software systems as we discussed in the previous chapters. Our visualization approach differs from existing software visualization techniques in two aspects:

- Adding an additional dimension comparing to the majority of two dimensional tools.
- Enriching the visualization with program analysis while reducing visual complexity due to information filtering and grouping.

For example, in Figure 25, we see JVC showing an example of a virtual city. It visualizes classes as buildings. The layout correlates with the coupling relationships among classes. This view permits us to answer questions about the size of the system, about the location (relative to other classes) and the size of classes.

Figure 25: An example of a Virtual City

The visualization approach is part of the CONCEPT (Comprehension Of Net-CEntered Programs and Techniques) [RILL02b] research project. The CONCEPT project is a reverse engineering environment that integrates software visualization and program analyses techniques, such as program slicing, design recovery [ZHAN03], and software metrics [WENG03]. Figure 26 [WENG03] shows an architectural overview of the CONCEPT project. In the following sections we discuss some of the implementation details including the integration of the JVC tool within the CONCEPT project.

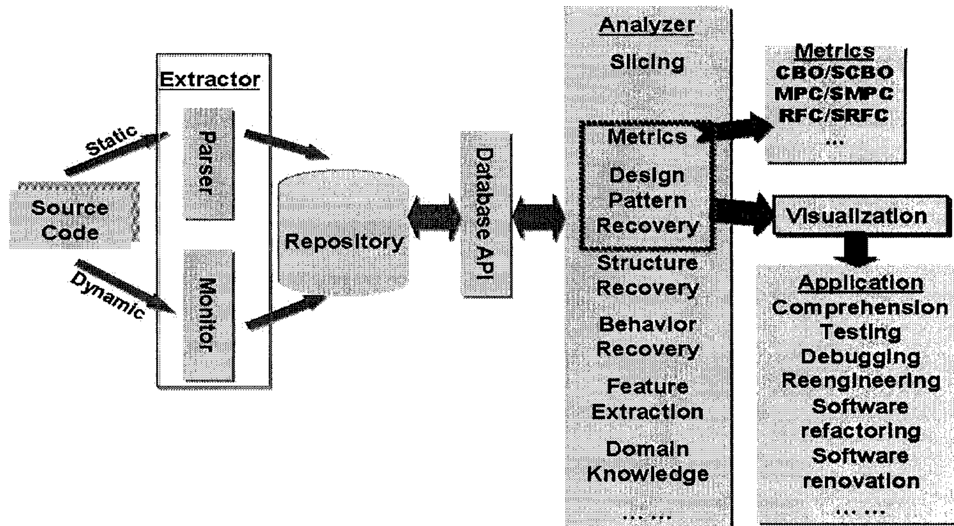

Figure 26: The Architecture of CONCEPT

4.2.2. The CONCEPT Project

The CONCEPT [RILLO2b] is a lightweight reverse engineering environment in which we utilize to investigate novel program comprehension techniques and approaches to assist programmers during the creation of mental models while comprehending software systems. Within our CONCEPT project, we are exploring new program slicing algorithms and their application in different software engineering sub-domain, e.g. software measurement, design pattern recovery, software visualization, feature analysis, and architectural recovery, etc.

The CONCEPT framework is built as a layered architecture as shown in Figure 26. The meta-model stores both static and dynamic source code information derived from the parsing and monitoring layer. The database API layer decouples the analysis layer (slicing, measurement, and feature extraction, etc) from the repository. The visualization and application layer are created on top of the analysis layer.

4.2.3. The Visualization Process

Our visualization process to illustrate our program representations is depicted in Figure 27. Source code is being read in during the Low-level analysis phase and stored as a source and class representation. While the source representation holds basically only an abstract syntax tree for syntactical program check, the class representation can be of different forms. Subsequently, high level analysis is conducted on the class representation in order to detect aspects, design, connectors and components, etc. The final information is stored within the aspect and the architecture representation.

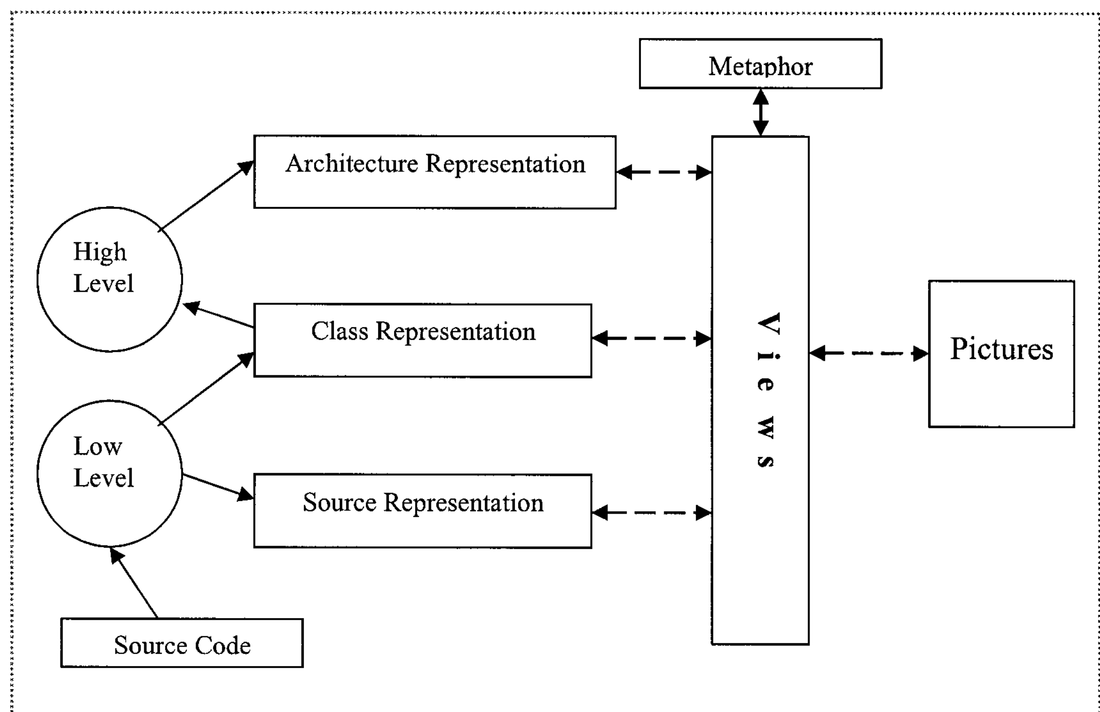

Figure 27: The Visualization Process in JVC

There are essentially three distinct conceptual parts to our JVC design. The first is the *GUI model,* which is an abstract model for the interface system. The second is the

3D graphics model, which is an abstract model for computing the 3D graphics. The final one is the *visualization model*, which is a data-flow model of the visualization process.

4.2.4.The GUI Model

The GUI model takes advantages of JFC/Swing's capabilities to realize multiple-view, user control etc functionalities. In Figure 28, the main window of JVC illustrates these concepts in a concrete form.

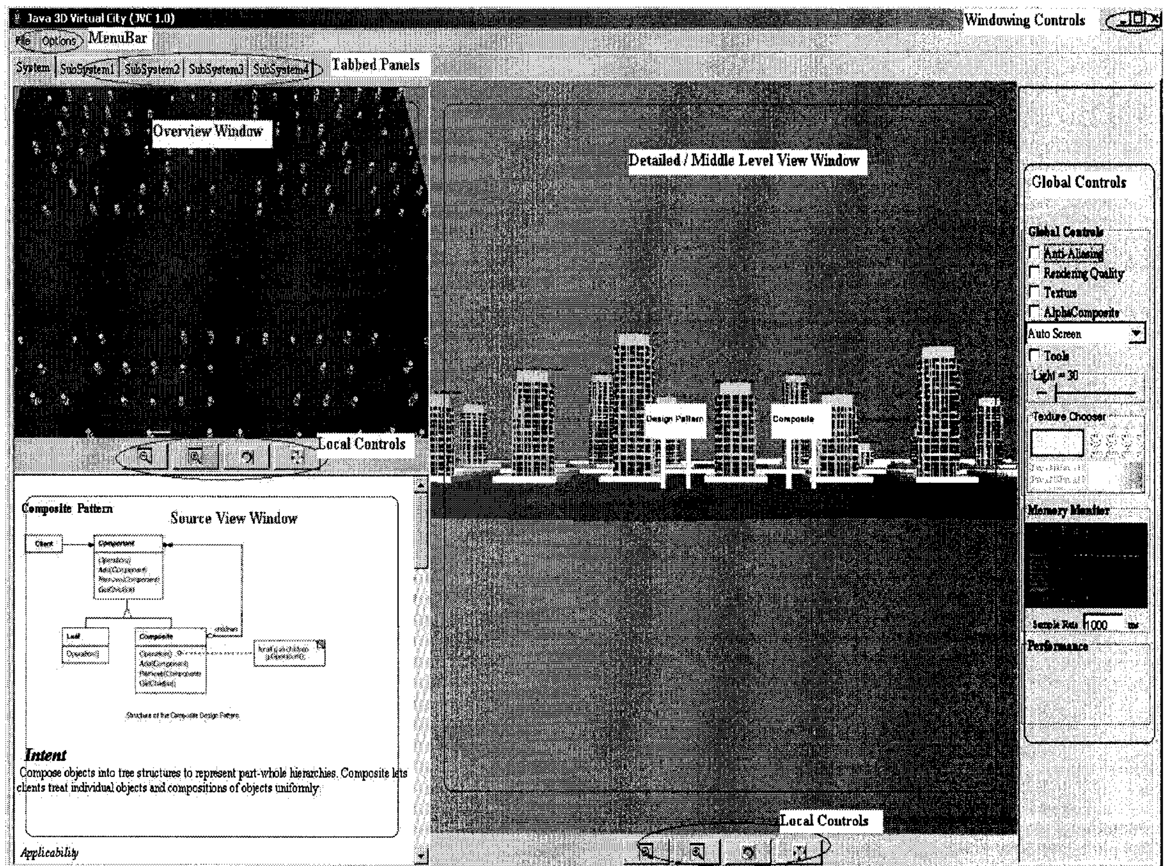

Figure 28: The main window of JVC

The GUI model contains some key objects as labeled in Figure 28:

Multiple Windows

The multiple-window design realizes the idea of program information residing in deferent levels as we stated earlier. Thus we implement three windows in the main interface: overview window, middle level view widow, and source view window. The overview window gives the user an overview about the target system; the middle level window displays information that the user is interested in. This is the major window where the user manipulates the scene; finally, the source window links to source code or its documentation, e.g. Java Doc. This window is powerful since it can display not only plain text file but also HTML file with the ability to trigger hyperlinks.

Tabbed Panels

The tabbed panel design is filtering technique that allows several components to share the same space. The user chooses which component to view by selecting the tab corresponding to the desired component. The system can automatically generate certain number of tabbed panels with tab titles according to the underline data.

Global and Local Controls

Both controls give users flexibility to manipulate the scene thus aid to establish their mental model of their interests. As the name suggested, global controls are functions being in charge of the whole system. We provide the graphic options such

as anti-aliasing, background changing, lighting etc. as well as some monitoring functions, for example, real time memory usage monitor.

Menu Bar

Menu bar is necessary to every application. In our system, the menu bar is an alternative way of some global controls and windowing functions. Further enhancement should be made to provide more functionality to the user.

Windowing Controls

Same as the most of windows applications, these controls allow the user to minimize, restore, change size of, and close the main window.

4.2.5. The 3D Graphics Model

The graphics model used for our JVC tool is based on Java3D scene graph, which is an acyclic, directed graph of nodes, where nodes correspond to such objects as primitive shapes, lights, cameras, viewpoints, transform groups, and branch groups. This model closely follows the state-machine based, graphics engine that is implemented in OpenGL. The rendering process is a traversal of the graph, where each node affects the current state of the rendering process. Thus, the order of the nodes in the graph has significant impact on the final image. Although the state-based traversal is powerful and efficient, it does violate a fundamental principle of object-oriented design. That is, the behavior of every object is completely determined from its inputs and local instance variables. In a scene graph, changes to a node in the graph can affect objects downstream of the graph traversal. Also, the Java3D scene

graph model is not intuitive to programmers who are attached to OO programming, like us. These are some of the reasons why it took us pretty efforts to master Java3D at the beginning.

Figure 29 gives an example of a typical scene graph to construct a virtual city scene in JVC. The VirtualCityPanel is a *JPanel* container which contains a *Simple VirtualUniverse*, which represents all the virtual objects that can exist within the particular 3D environment; a *Canvas3D*, which provides a drawing canvas for 3D rendering; a *ViewPlatform*; which controls the position, orientation, and scale of the viewer; and the city *BranchGroup,* which serves as the only pointer to the root of the whole scene graph by being attached to the *Local* object in the *Simple VirtualUniverse.*

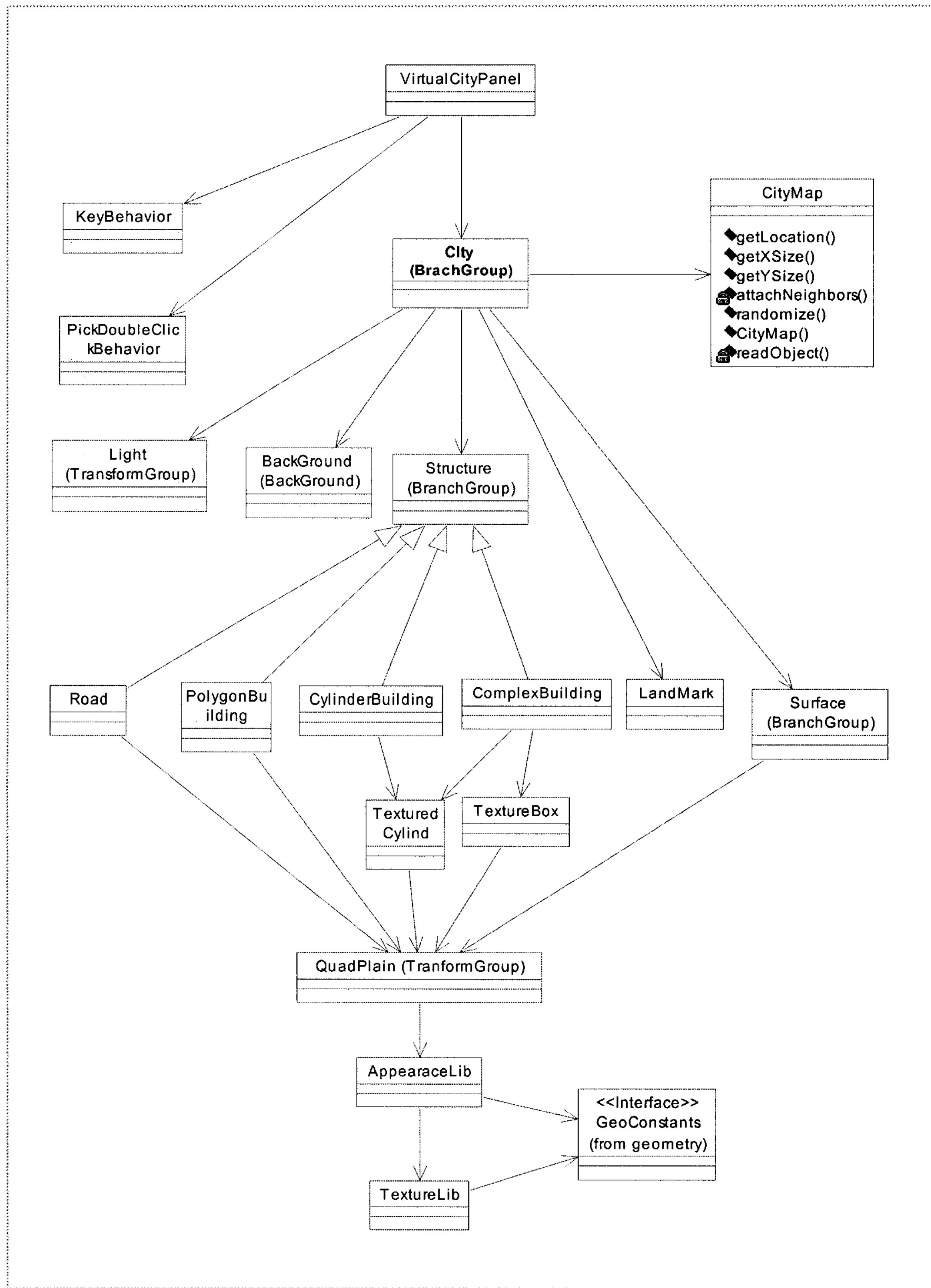

Figure 29: Class diagram for a typical scene graph in JVC

4.2.6. The Visualization Model

The visualization model captures the essential of the visualization tool. It is based on the data-flow paradigm adopted by many commercial systems. Figure 30 gives a detailed view of dataflow within the whole CONCEPT environment, where our visualization model corresponds to the *Visualization Processing* segment.

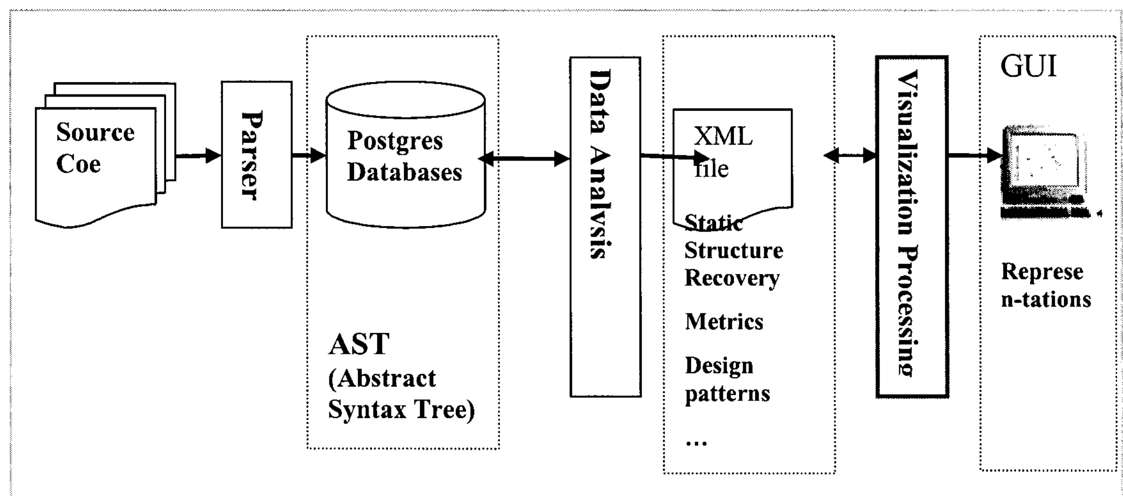

Figure 30: The detailed dataflow within the CONCEPT

In the dataflow paradigm, modules in the visualization model are connected together into a chain. The modules perform algorithmic operations on data as it flows through the chain. The execution of visualization chain is controlled in response to demands for data (demand-driven) or in response to user input (event driven). The appeal of this model is that it is flexible, and can be quickly adapted to different data types or new algorithmic implementations.

Our visualization model (see Figure 31) consists of two basic types of objects: *process objects* and *data objects*. Process objects are the modules, or algorithmic

portions of the visualization chain. Data objects, also referred to as datasets, represent and enable operations on the data that flows through the chain.

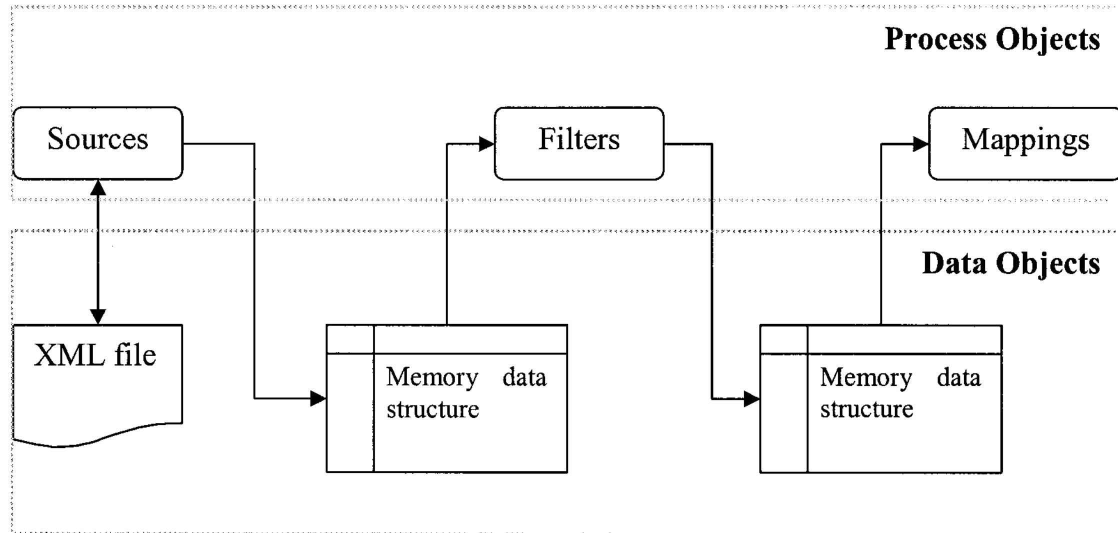

Figure 31: JVC Visualization Model

Process objects may be further classified into one of three types: *source, filter, and mapping* objects (see Figure 31). Source objects initiate the chain by reading an XML data file, generate one or more output datasets, and write or update the data file. (Figure 32 is the XML pipeline in JVC). Filters take one or more inputs and generate one or more outputs. Mappings, which require one or more inputs, terminate the chain.

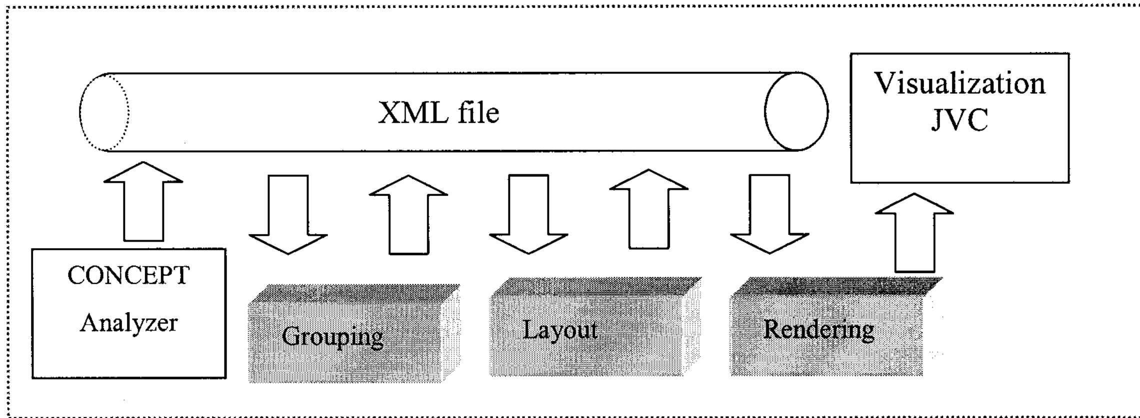

Figure 32: The XML pipeline in JVC

In order to represent the map of a virtual city, we choose a structured *Grid,* composed by *Cell* and *Line,* as the major data type in JVC. The class diagram for this data model is shown in Figure 33. An important feature of this data model is the concept of cells. A grid consists of one or more cells according to size of objects to be visualized. Each cell is considered to be atomic visualization primitive. Cells represent topological relationships between the points that compose the dataset. The primary function of cells is to locally interpolate data or compute derivatives of data. Further, a cell can be a divided in to a sub-grid. This feature enables the visualization model can deal with any number of objects. Lines represent relationships between cells. We use HillClimb algorithm to minimize line crossing in the map as shown in Figure 33. We introduce the grid layout and grouping algorithm in section 4.3.2.

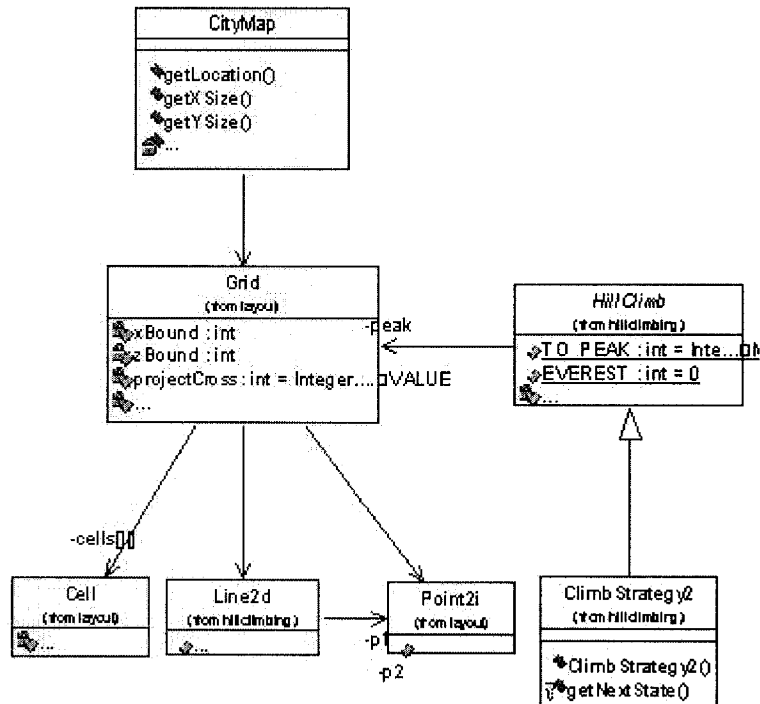

Figure 33 : The class diagram of JVC data model

4.3 Implementation Issues

4.3.1. The System Structure of JVC

The system structure of JVC is shown in Figure 34. There are many challenges in implementing this structure. One of the most is mixing the *lightweight* Java Swing and the *heavyweight* Java3D components. A *heavyweight* component is one that is associated with its own native screen resource (commonly known as a *peer*). A *lightweight* component is one that "borrows" the screen resource of an ancestor (which means it has no native resource of its own -- so it's "lighter").

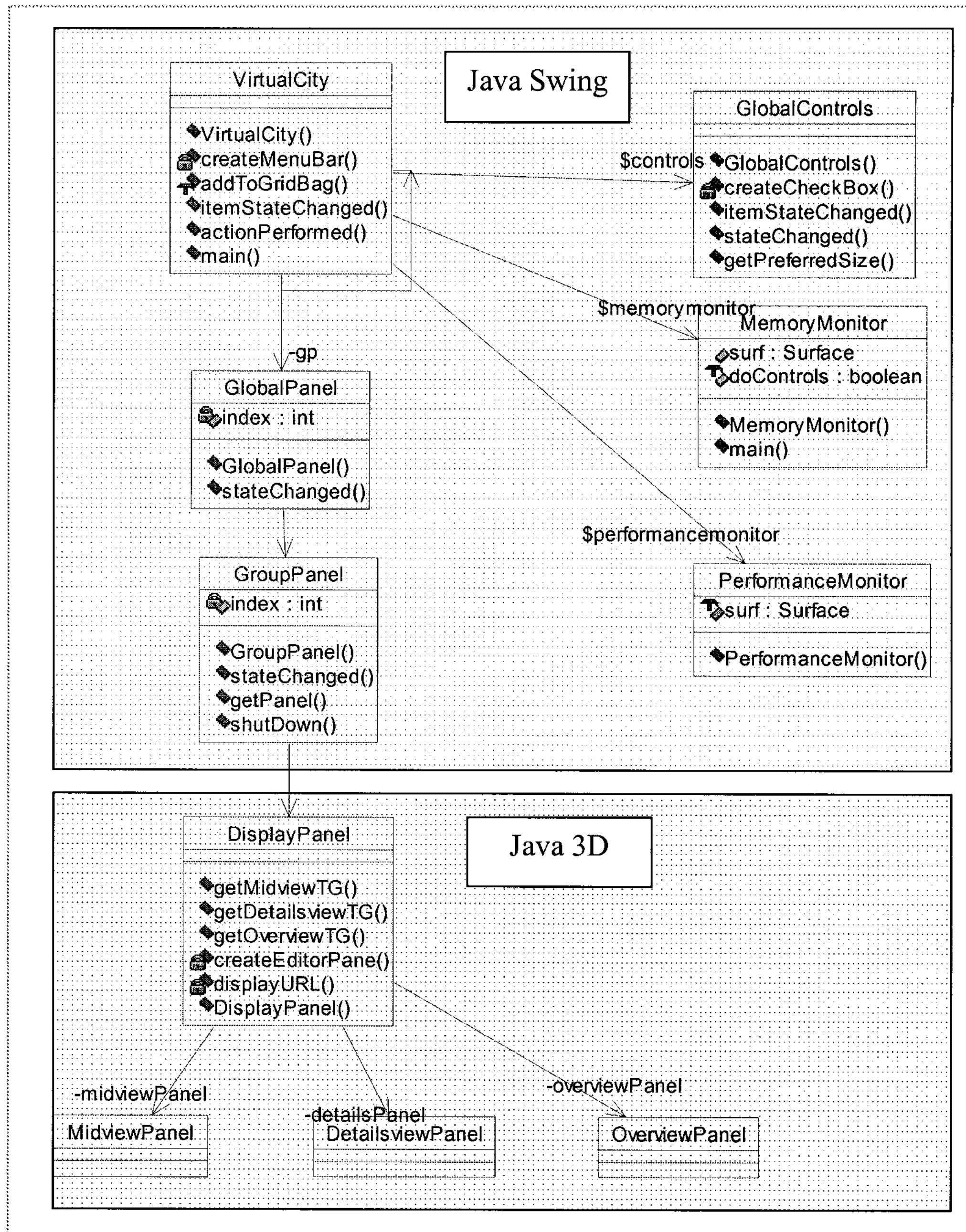

Figure 34 Class diagram for JVC system structure

To integrate Java3D and Swing, The basic trick is to add a *Canvas3D* to a *JPanel* so that a scene can be put in a GUI as needed. This means that a *Canvas3D* will draw on top of Swing objects no matter what order Swing thinks it should draw in, thus

causes some problems, for example, the Swing pop-up menus do not work properly with Java3D, if the menus were not forced to change to *heavyweight* from its default *lightweight*. The integration between Swing and Java3D is not perfect. There are overdraw issues and texture management issues. With careful design, most of these problems can be worked around.

4.3.2. Layout and Grouping

The graph layout problem is an ongoing research area within the information visualization domain for the last decades. One of the reasons is that many layout algorithms are not scalable. "Few systems can claim to deal effectively with thousands of nodes, although graphs with this order of magnitude appear in a wide variety of applications (HERM00)." A layout algorithm working well for a small system could become completely useless when the size of the system increases. Another reason is that many algorithms are lack of full automation. In this context, Eades and Xeumin [EADE89] suggest three general criteria that constitute "good" drawing of directed graphs:

1) Avoid upward pointing arcs;

2) Distribute nodes evenly across the screen;

3) Minimize arc crossings.

Grid Layout

The *Grid Layout* algorithm implemented in JVC is scalable and automatic while trying to meet the three criteria. The automatic layout algorithm contains three basic steps.

Step 1: Determine the number of nodes and the size of the grid. The grid size is corresponding to the number of nodes to be put. Later the initial position of the viewpoint is determined by the size of the grid.

Step 2: Construct the grid and randomly lay out each node. This is accomplished in two stages:

a) Construct the "neighborhood" relationships of each cell in the grid. This is a must for the *Hill Climbing* algorithm in JVC.

b) Assign nodes to cells based on a random mechanism.

Step 3: Optimize the layout to minimize line crossing by using the *Hill Climbing* algorithm.

The third step is optional. When discarding this step, the computational complexity of the grid layout algorithm is O(n) where n is the number of the nodes to be displayed. In what follows, we discuss the Hill Climbing algorithm used for the optimization in step 3.

Hill Climbing Algorithm for Cross Minimizing

The basic idea of *Hill Climbing* is to always head towards a state which is better than the current one. The general steps of this algorithm can be described as follows:

1. Pick a random point in the search space

2. Consider all the neighbors of the current state

3. Choose the neighbor with the best quality and move to that state

4. Repeat 2 thru 4 until all the neighboring states are of lower quality

5. Return the current state as the solution state

More details about the algorithm can be found in many *Artificial Intelligence* literatures.

In this thesis research, we reuse an implementation [WANG03] developed by Wang within the CONCEPT research. In this implementation, Hill climbing discards the states which is worst than the current one, thus its *space complexity* is O(1). The computation terminates when there are no successors of the current state which are better than the current state itself. In this implementation, its *time complexity* is $O(n^2)$ where n is the number of the entities in the grid.

In what follows we explain how this algorithm can minimize line crossing by taking coupling relations as parameters.

A coupling data can be described as: $C(x_1, x_2, c)$, which can be drawn as a line X from the position of x_1 to the position of x_2 with weight c. Let the optimization decision variable be $X = f(X_1, X_2, \ldots, X_n)$, where f is the function summing the values of line crossing among the lines $X_1, X_2, \ldots, X_n$. So we apply the Hill Climbing algorithm in steps as shown in Figure 35. The basic idea is that continuously swap position a random x_i with one of its neighbors until it gets a lowest value of the decision variable or reaches the peak.

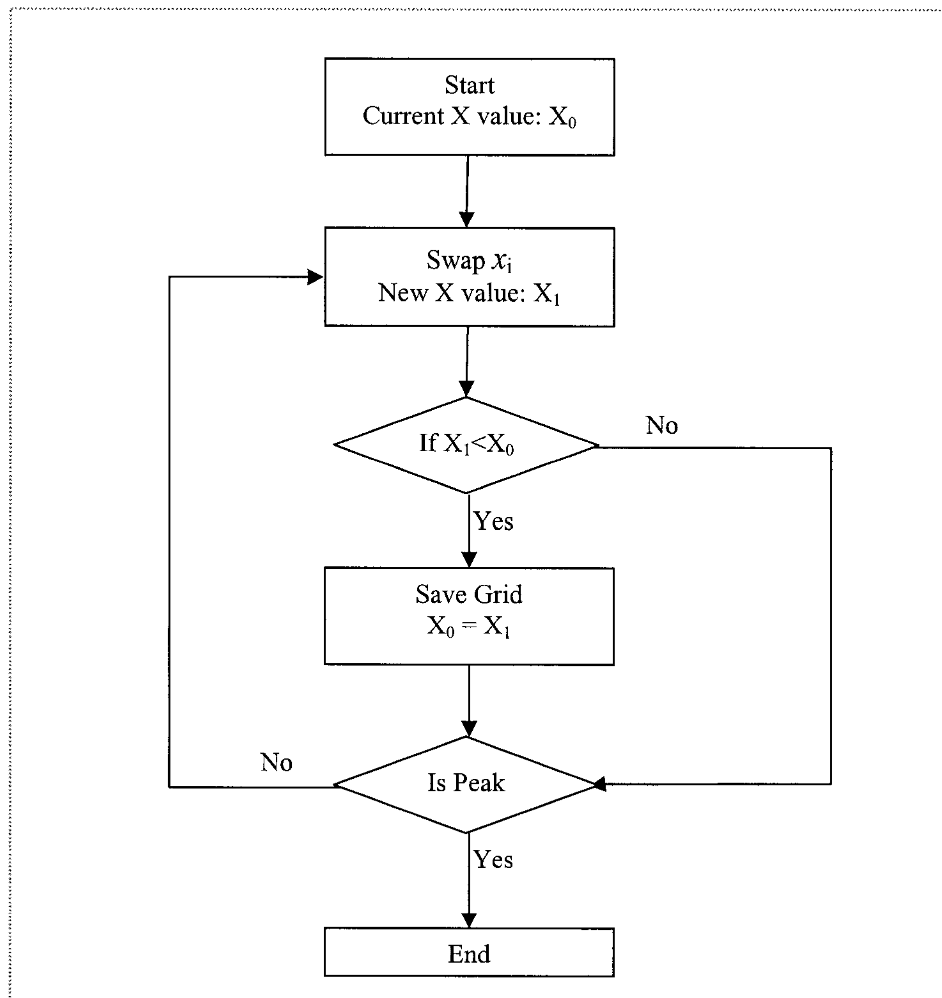

Figure 35: Control flow of Hill Climbing in JVC

Grouping

It is clear that layout algorithms on their own often do not provide enough insights and details to be useful for comprehension tasks. The layout algorithms are constrained by the amount of information to be displayed and the limited screen space. Even, if one manages to create a layout, the resulting visual might have far too much information, causing an information overload. Furthermore, layout algorithms are in general only concerned with minimizing the crossings among entities. However, it has to be noted that minimizing the crossings, might not necessarily correspond to the logical view a programmer had. Therefore, limiting the number of entities to be displayed and their logical organization to match the user's mental model of the system is one of the key challenges in software visualization. For the visualization of large software systems, it is essential to provide some type of grouping to create a decomposition of the system. The grouping can improve readability by supporting a representation that is closely related to the mental model.

As shown in Figure 36, we can reserve an area of cells easily by setting their *Boolean* property: *is empty* or not. Or we can group some nodes together by putting them in a split cell as also shown in Figure 36.

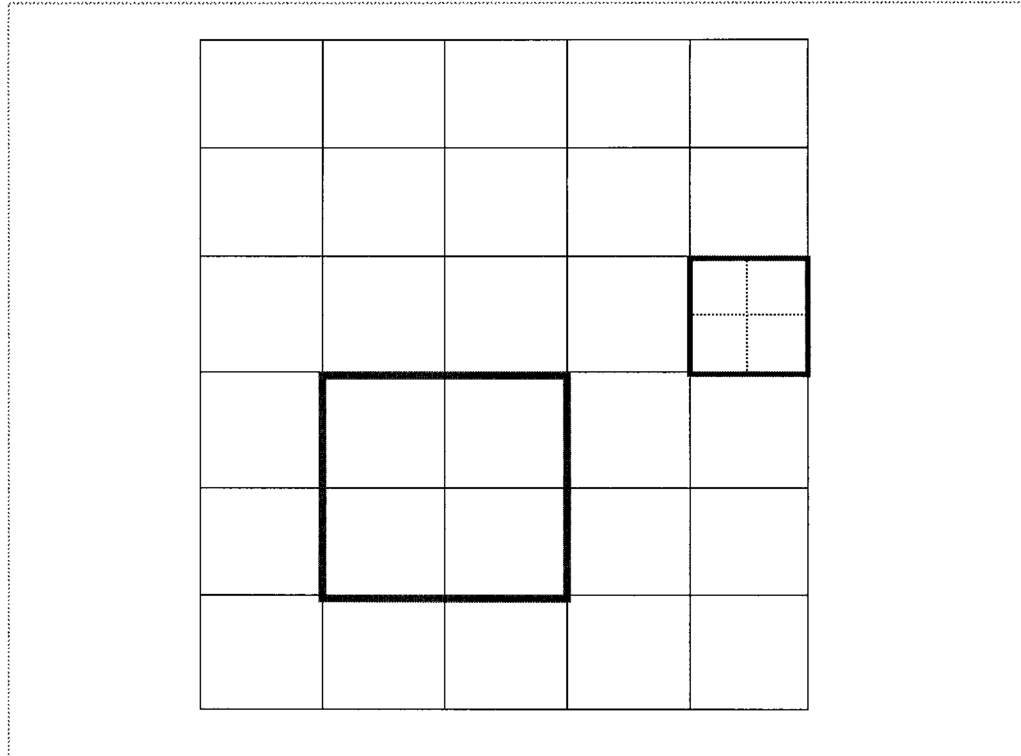

Figure 36: Group and split cells in a grid

4.3.3. Navigation Methods

The main purpose of visualizations is to facilitate the exploration of information. Visualizations therefore have to provide navigation tools. Associated with navigation, two concepts, *global context* and *local focus*, are very important.

Global Context

It is very important for the users to quickly gain a rough overview of the structure of the system. This helps users to build a representation of this structure in their mind. Such a generated global context gives users a *feeling* for the structure (see example in Figure 37), so users know approximately where to find some

specific information. Or which parts of the components are very large and therefore contain much information. This contextual information obviously speeds up navigation through the information space. The 3D representation of the Virtual City provides the user with multiple views: overview with highlighting focus, related source view, and detailed view about the focus.

Navigating through the visualization helps also to build contextual information, but it must always be apparent to the users where they move. That means moving from one position to another should be smoothly animated. This is usual in interactive navigation modes, where the users determine which direction and which speed they want.

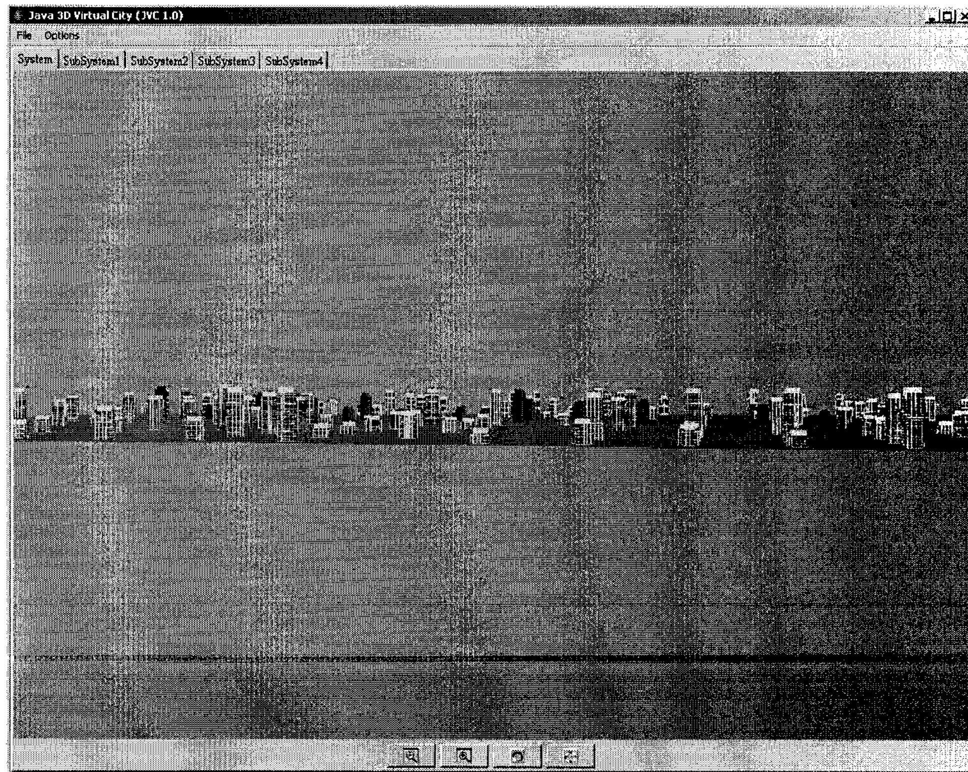

Figure 37: A global view

The left top window provides a global overview. Looking down on the landscape from a position right above the Virtual City, results in a top view of the city. This view is very meaningful, because it easily can be seen how the classes are structured and distributed

Local Focus

It is not only important to get a feeling for the overall structure of the system. Usually, users want to quickly find the specific interests. After that they usually want to focus on the specific components to do some work. Thus only a fraction of the system is displayed and therefore the frame rate will be easier to maintain at interactive rates (see example in Figure **38**).

Figure 38: A detailed view

Corresponding to these concepts, many different navigation methods are implemented in the JVC. However, the input devices of an average PC are not designed for use with a three-dimensional interface. So we have to put great efforts in design a user interface and mouse and keyboard behaviors for the 3D visualization. Such navigation aids should include:

- A navigation mode to move freely through the 3D space. Such a navigation mode ensures the user gains an overview over the hierarchic structure.
- The user should be able to easily navigate to the interesting information. This means providing mechanisms to move easily to child nodes or to the parent node, so the user navigates rapidly to the desired level of the hierarchy.
- Special views of the cityscape such as a top view should be reachable quickly. A top view displays the ground plan of a plateau. This should not only work for the root plateau, but also for every other plateau.
- Often used views of the hierarchy should be reached without much navigation. A simple mouse click or a key shortcut should be enough to move to such viewpoints of the system. Also, facilities to define and mange such viewpoints should be provided.
- Search functionality can be used to navigate quickly through the information space. The search results could be highlighted and some easy to use facility should move the user between these objects.

- A navigation history which tracks the navigation through the 3D space. It would be useful to be able to undo or redo navigational actions.

We provide users traditional mouse and keyboard method to zoom in/out, rotate, and fly/walk through the scene as well as buttons to realize the same functions. As the input device technology advances, we introduce a new device: data glove.

Data Glove

The data glove is an input device for *virtual reality* in the form of a glove that measures the movements of the wearer's fingers and transmits them to the computer. Sophisticated data gloves also measure movement of the wrist and elbow. A data glove may also contain control buttons or act as an output device, e.g. vibrating under control of the computer. The user usually sees a virtual image of the data glove and can point or grip and push objects.

The data glove provides an intuitive "grab-and-drag" metaphor. In Figure 39, the user wears a wireless data glove. When the user bends all fingers into a fist, the entire world can be moved by moving the hand. As long the hand is not in a fist, the user can manipulate other types of interactions with the virtual world. The data glove navigation plug-in is currently developed by other members of the CONCEPT project group. Once the plug-in is available we will integrate this navigation technique in our JVC tool.

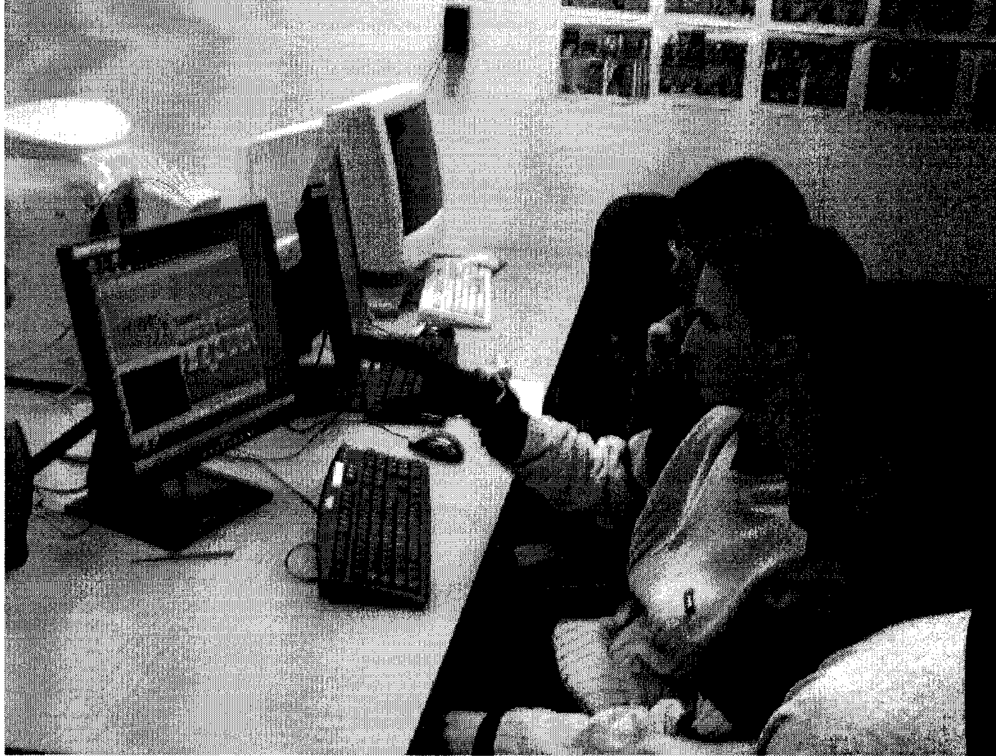

Figure 39: Navigate in JVC using data glove

4.3.4. Memory Usage

This issue is important because of some reasons stated as following. First, the data-flow form consumes a lot memory. Second, Java is a high memory consumption language, especially for textures. Java3D will make internal copies of the data so that it is not affected by application code making changes to it. We can skip this intermediate copy by using the BY_REFERENCE capabilities. But this is still not enough to deal with the memory problem. As mentioned before, the textures consume huge amount of memory even using BY_REFERENCE. We solve this problem by building a static texture "bank" in memory.

Chapter 5

Application of JVC: Visualizing Design Pattern

In this section, we provide a case study showing the application of our JVC environment for the comprehension of software systems. The essence of program comprehension is in identifying program artifacts and understanding their relationships. This process is essentially pattern matching at various abstraction levels via mental pattern recognition by the software engineer and the aggregation of these artifacts to form more abstract system representations. In this context, the comprehension of source code plays a predominant role in ensuring quality during software maintenance and evolution. One approach to improve the comprehension of software systems through reverse engineering is by providing higher level of abstraction through visualizing the data that has to be observed and inspected.

Design patterns are typically modeled in UML. However, the application and implementation of design patterns and the role of the design pattern might get lost. This is a major compromise on the benefits of design patterns, because the domain knowledge and design decisions associated with the original patterns are no longer available. In this context, tools that enable the developer and maintainer to recognize and visualize abstractions at a higher level than the code provide an excellent vehicle for understanding the higher level design and design intents.

In this section, we describe the experiment of using the JVC tool to visualize design patterns recovered from the CONCEPT design pattern recovery process [ZHAN03].

5.1 Introduction to Design Patterns

Design patterns have become increasingly popular among software developers since the early 1990s. The culmination of early technical discussions was the publication, *Design Patterns -- Elements of Reusable Software*, by Gamma, Helm, Johnson and Vlissides (1994) [GAMM 94]. This book, commonly referred to as the Gang of Four or "GoF" book, has had a powerful impact on those seeking to understand how to use design patterns. Some useful definitions of design patterns have emerged as the literature in this field has expanded:

- "Design patterns are recurring solutions to design problems you see over" [ALPE 98].
- "Design patterns constitute a set of rules describing how to accomplish certain tasks in the realm of software development." [PREE 94]
- "Design patterns focus more on reuse of recurring architectural design themes, while frameworks focus on detailed design… and implementation." [COPL 95]
- "A pattern addresses a recurring design problem that arises in specific design situations and presents a solution to it" [BUSH 96]

- "Patterns identify and specify abstractions that are above the level of single classes and instances, or of components." [GAMM 93]

From these definitions, we can see that design patterns are not only about the design of objects, but also about the communication between objects.

Design patterns can exist at many levels from very low level specific solutions to broadly generalized system issues. There are now in fact hundreds of patterns in the literature. They have been discussed in articles and at conferences of all levels of granularity.

Table 6 shows the 23 design patterns selected for inclusion in the original "GoF" book. They are on a middle level of generality, where they could easily cross application areas and encompass several objects. The authors divided these patterns into three types creational, structural and behavioral.

- **Creational patterns** concern the process of object creation. This gives programs more flexibility in deciding which objects need to be created for a given case.

- **Structural patterns** deal with the composition of classes or objects. This helps to compose groups of objects into larger structures, such as complex user interfaces or accounting data.

- **Behavioral patterns** characterize the ways in which classes or objects interact and distribute responsibility. This helps to define the communication between objects in a system and how the flow is controlled in a complex program.

Table 6 Design pattern space

	Purpose			
		Creational	**Structural**	**Behavioral**
Scope	Class	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

5.2 Design Pattern Recovery

Design patterns [GAMM 94] 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". Reverse engineering focuses on creating "representations of a system in another form at a higher level of abstraction" [CAMP 97].

5.2.1. Literature Review

Many researchers have presented their efforts on automatic recovery of design patterns in large object oriented programs. As we shall discuss below, most of these adopt the approach of describing patterns at some level of formalism and then match the descriptions with the facts extracted from source code.

The Pat [KRAM 96] system is a Reverse Engineering tool that searches for design pattern instances in existing software. Within Pat, design information is "extracted directly from C++ header files and stored in a repository" [KRAM 96]; "the patterns are expressed as PROLOG rules and the design information is then used to search for all patterns" [KRAM 96]. Since the Pat system is designed to collect information just from C++ header files without semantic analysis of the method body, the authors limit the considered patterns to five structural patterns introduced in [GAMM 94]: Adapter, Bridge, Composite, Decorator, and Proxy. The authors of the Pat system state that in the 4 benchmark applications, the precision ranges from 14 to 50 percent, and will be "much higher if Pat could also check for correct method call delegations" [KRAM96].

In [ANTO98], Antoniol et al. present a more conservative approach to recover design patterns from design and source code. Their approach is mainly based on "a multi-stage reduction strategy using software metrics and structural properties to extract structural patterns for OO design or code" [ANTO98]. Similar to the Pat system, this approach also focuses on the same five structural patterns mentioned in [KRAM96]. However, in addition to Pat, method delegation a method delegates its responsibilities by calling another method of associated class is used as a design pattern constraint to reduce the reported false candidates. The authors conclude that in public domain code case studies the average precision is 55 %, and that there is "an increase of about 35% using also the delegation constraint with respect to the use of structural constraints alone" [ANTO98].

In [FERE01], Ferenc et al. state that Columbus and Maisa pairs, two reverse engineering tools, can be used to document and analyze software implemented in C++ and to verify the architectural design decisions during the software implementation phase as well. The authors conclude that some patterns, like Iterator and Observer, cannot be recognized with the current method because the definitions of these patterns contain generated facts, i.e., "structural facts that are dynamically pushed to the input by Maisa when it recognizes a particular kind of pattern or a special kind of a common class relation". As well as this limitation, they also report that the performance degrades with large software systems [FERE01].

In [HEUZ03], Dirk Heuzeroth et al. argue that static analyses are not sufficient for pattern recovery, and they therefore introduce dynamic analysis techniques to detect design patterns in legacy code. Through their approach, a set of potential patterns are firstly detected by a static semantic analysis tool. These candidates are then examined by dynamic analysis, which are performed by tracing program executions. The authors report a tremendous improvement in the recovery quality – the number of false positive cases is very small and "in most experiments even zero" [HEUZ03].

5.2.2. Design Pattern Recovery in CONCEPT

In this section, we briefly introduce our approach to the recovery [ZHAN03] of the 23 patterns introduced in [GAMM94] from Java source code. The presented approach is based on the theoretical foundations of LePUS [EDEN02], a formal

specification language dedicated to the general OO design and architecture and has been implemented by [ZHAN03] within the CONCEPT project.

In the LePUS language, the solution part of design pattern can be described as a set of participants, i.e. classes and methods, and their collaborations. In order to formalize the participants of a design pattern, LePUS defines Class and Method as ground entities, and further, uniform sets and higher order sets as a set of participants. Clan and Hierarchies are also introduced in order to represent the dynamic binding and inheritance mechanisms of OO programming respectively. In order to formalize the collaborations of participants, LePUS identifies a small set of relations between ground entities, like DefineIn, ReturnType, etc., as ground relations. Set relations and function can also be applied.

In our implementation, we use a simplified version of LePUS to formalize design patterns, which eliminates some high-order concepts to reduce the complexity of the formalism. At the same time, we introduce additional relations to extend the expressiveness of the language. As a result, each pattern is formalized in several formulae that can be easily implemented.

The recovery process starts from static analysis of Java source code. A source code analysis tool is adopted to semantically identify ground entities and relations from source code. Our pattern recovery tool matches the definition of pattern and the extracted information to identify design patterns in the program.

The approach has been fully implemented and can be used to identify several design patterns listed in [GAMM94]. We consider our approach as conservative –

that is, all segments of source code which match the definition of pattern will be reported. However, some false positive pattern recognitions still occur. This occurs in situations when a source code artifact has the same structure of that of the pattern definition, but a human with the necessary design pattern domain knowledge will be able to recognize that this particular design was not intended to be a design pattern in that particular context.

5.3 Visualizing Design patterns

5.3.1.Target System

Our test is performed to a medium size system written in Java, which has 201 classes. After performing some our CONCEPT program analysis techniques, we got higher level information. Through software metric measurement [MENG03], we got 599 coupling relations among these classes. Through design pattern recovery [ZHAN03], we discovered 67 "GoF" design patterns in 8 kinds with 1205 participants (classes, methods, and attributes).

5.3.2. The Input XML File

All these information are written into one XML file. Figure 40 lists a short part of this data file as an example. The file contains three separate kinds of information: Class information, Coupling relations, and Design patterns. As we can see in the sample file, class information includes class ID number, name, number of lines of code, number of method, etc. The coupling relationships are obtained by the CONCEPT metrics analysis [WENG03]. Each relationship contains a source class, a destination class, and a weight measurement of this relationship. Design patterns

are recovered by design pattern recovery techniques stated in 5.2.2. We numerate each design pattern as its unique ID (*patternID*). Each pattern data contains the pattern type name, participate entity ID (*componentID*), name (*componentName*), and hierarchy level (*componentType*), and its description.


```
<?xml version="1.0" encoding="UTF-8" ?>
- <entity_relation>
    <! – the following is class information -->
    - <entities>
        - <entity id="200" name="concept.java.tree.VarDeclarationStatement" numofline="98", numofmethod="5"> <coordinator x="0" y="0" z="0" />
        </entity>
    </entities>
    <! – the following is coupling relationship data -->
    - <relationships>
        <relationship from="200" to="24" weight="1" />
        <relationship from="200" to="83" weight="1" />
    </relationships>
    <! – the following is design pattern data -->
    - <patterns>
        - <pattern patternID="11" patternName="Singleton"> <component componentID="0" componentName="concept.java.FileClassLoader" componentType="100" description="singleton class in Singleton pattern" />
        </pattern>
    </patterns>
</entity_relation>
```


Figure 40: Sample XML data in JVC

5.3.3. The Results

Using the city landscape to represent design patterns has shown to be a promising approach to visualize the structure of these patterns for several reasons. Using the city landscape provides a mental model that closely corresponds to the visualizations used by city planners to model and design city layouts

The height of the buildings maps to the size of the classes (relative to the source code). Color (texture) is used to indicate hotspots and to visually distinguish classes and their participation in design patterns. Figure 41 shows a screen capture view created by JVC. The left top window shows a global system view providing the user with a general overview of the system and its structure. The main window provides a detailed view which is dependant on the selected view. The user can select among several views and filtering techniques to reduce the information complexity. Within the Virtual City, we provide the user with the ability to focus either on the structure of a particular instance of a design pattern in the program (shown in Figure 42), while Figure 43 shows a screen shot of the Java Doc of the system. The left bottom window provides a description of the "GoF" description of the design pattern, including information about the intent and usage of the specific design pattern.

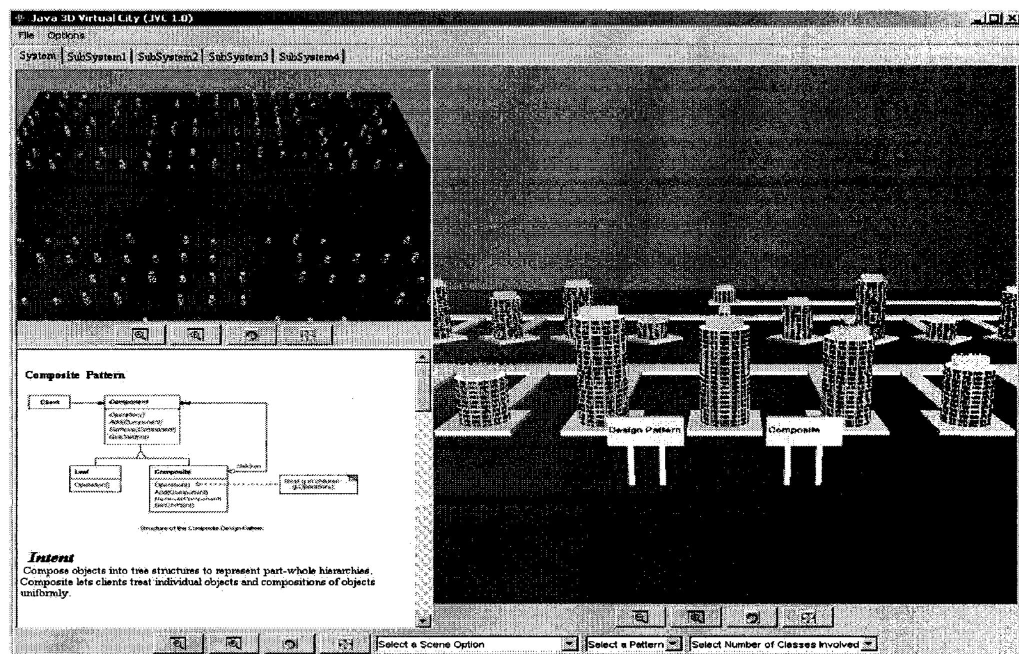

Figure 41: Design pattern visualization in JVC

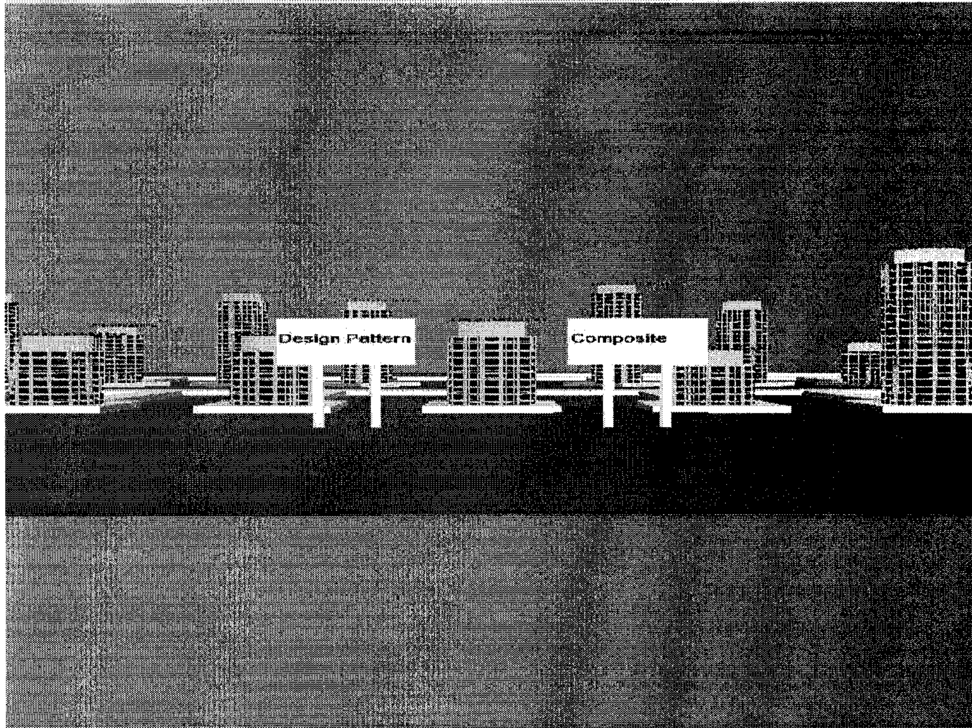

Figure 42: Close view of a design pattern

City - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Address C:\concept\virtualcity\doc\javadoc\index.html

RETURN
All Classes

Packages
<unnamed package>
algorithm.layout
algorithm.layout.astar
algorithm.layout.hillclimbi

All Classes
AStar
AStarTest
AnimateTexturesBehavio
AppearanceLib
BolckPlane
Building
BuildingEntity
CAppearance
CXml
Cell
Chair
CheckerFloor
City
ClimbStrategy2
ColorConstant
ColouredTiles
ComplexBuilding
CylinderBuilding
DatabaseServer
DatebaseServer
DepthFirstSearch
DesignPatternPanel
DetailsviewPanel
Disc

Overview Package Class Tree Deprecated Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class City

java.lang.Object
└City

public class City
extends java.lang.Object

Constructor Summary

City()

City(Grid map, javax.media.j3d.Canvas3D canvas)

Method Summary

void drawRelationLines()

int getBlockSize()

Figure 43: Java Doc of a class

Figure 44 shows two zoomed and rotated views of the design pattern providing detailed visuals on the different participants in the pattern and their roles.

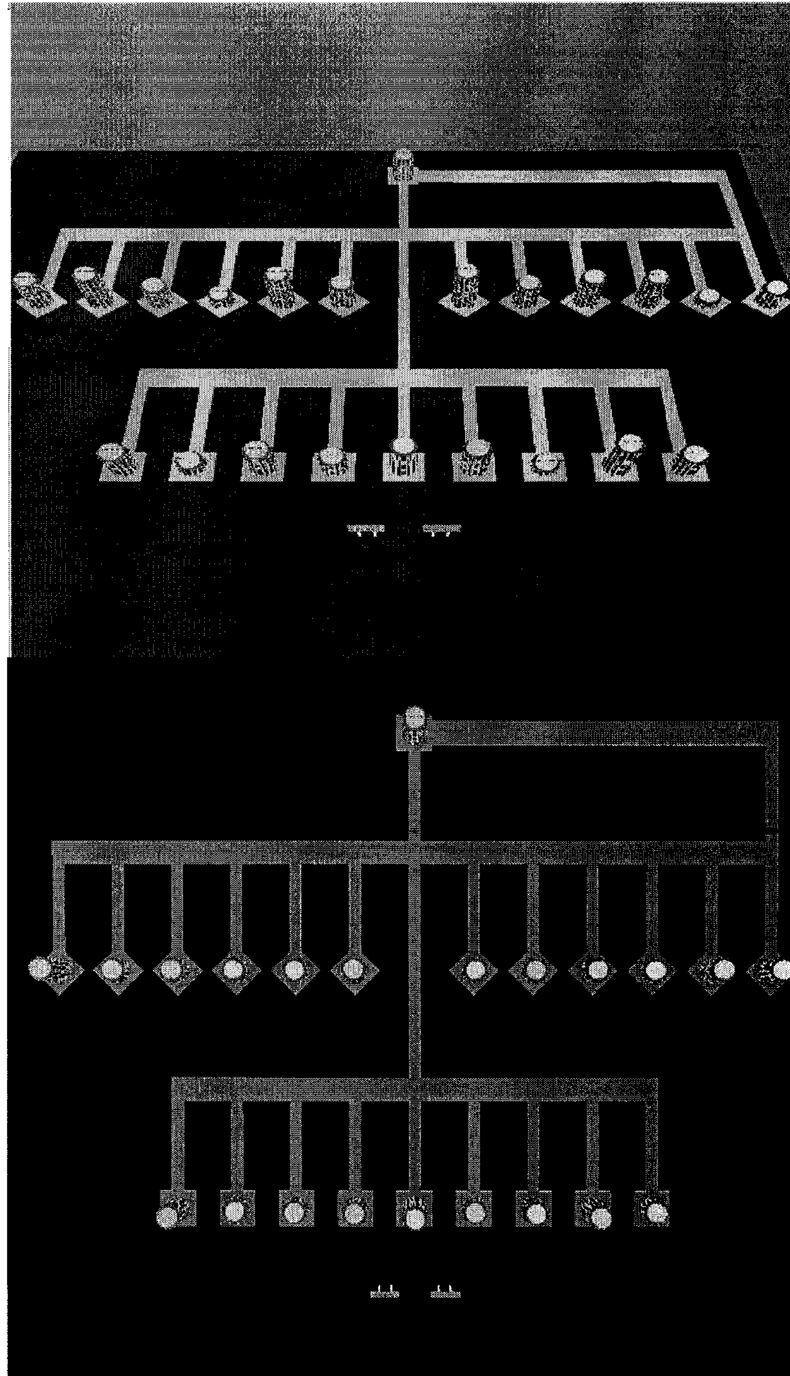

Figure 44: Detailed view of a design pattern

Other filtering options include the visualization of design patterns of a specific size (depending on the number of participating classes) or by selecting only design patterns of a specific type (see Figure 45). Figure 46 illustrates two views using filtering techniques to show only pattern classes or the other classes without pattern classes.

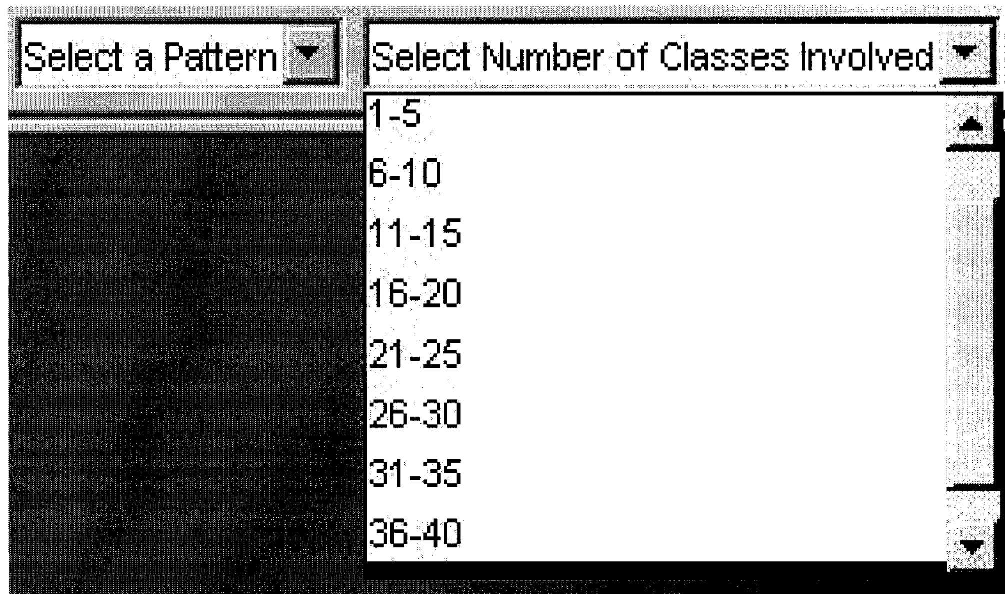

Figure 45: The ComboBox for selecting options

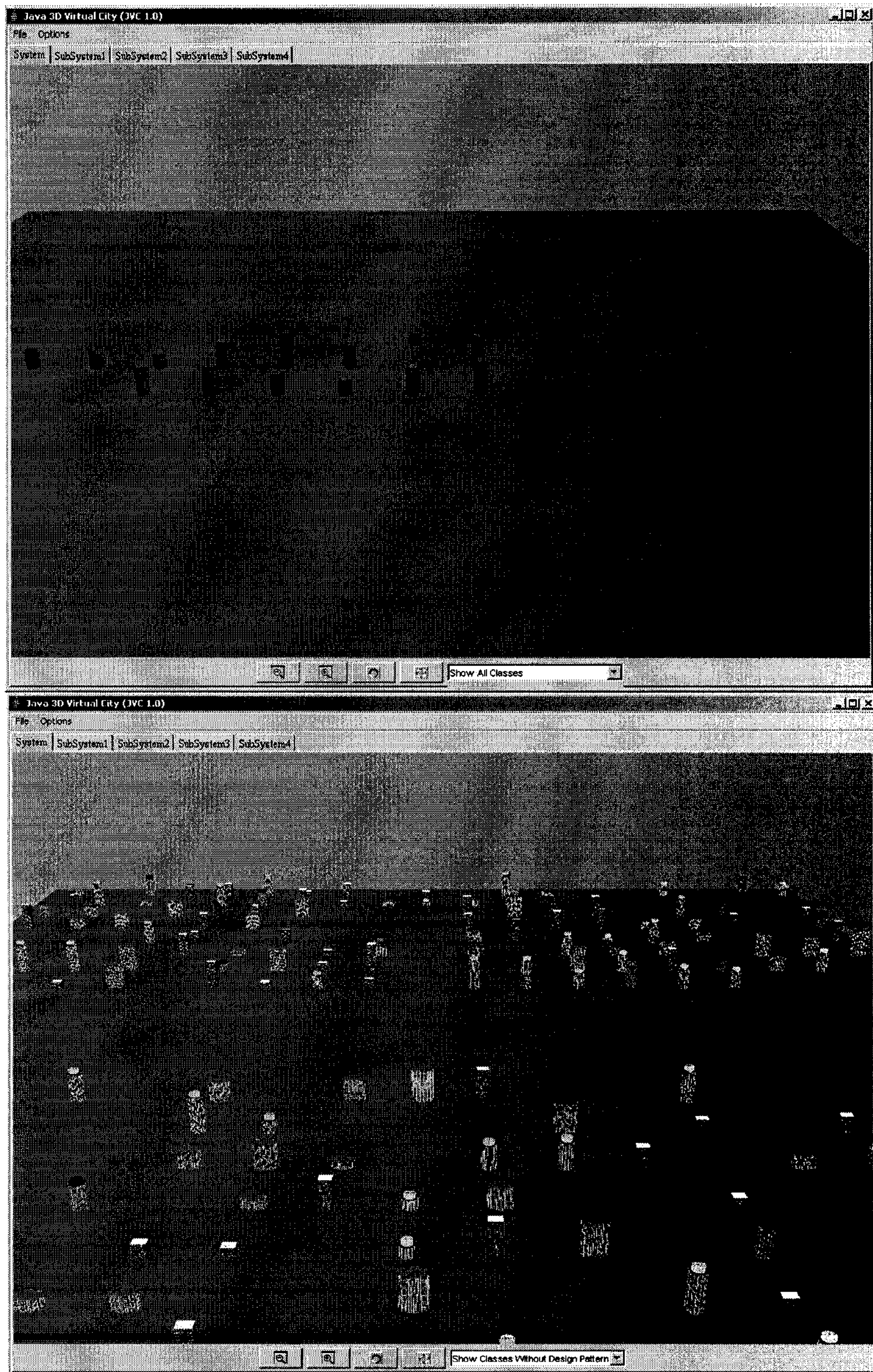

Figure 46: A virtual city shows pattern classes only (top) or classes without the pattern classes (bottom)

5.3.4.Discussions and Limitations

In our case study of visualizing design pattern, we illustrate the importance of providing visual abstractions that provide an intuitive representation of information derived from source code analysis. The presented case study presents the applicability of the presented visualization approach for a small-medium size system. The use of different abstraction levels improves the navigation through the virtual city representation. Filtering techniques provide an essential mechanism to reduce the amount of information and therefore the cognitive load for the user. However, our current approach is limited by several major factors. (1) Providing an intuitive and to the user meaningful visualization, it will require that the design pattern visualization closely matches the expected representation of the pattern (e.g. as documented in the "GoF" book). Achieving such a close representation however will require the provision of layout algorithms that are specialized in visualizing a particular design pattern. (2) We have not yet addressed the issue of crosscutting of design patterns. This situation commonly occurs when a class (or its components) participates more then in one design pattern. (3) The design pattern recovery process and its precision will have to further be optimized to reduce the number of false positive detections. This corresponds to situations, and then the algorithm wrongly detects patterns in the source code.

Chapter 6

Conclusions and Future Works

Numerous techniques to visualize software systems are available today. Several of these techniques go beyond the traditional approach of 2D representations and utilize the 3D space. In this research, we explored the Virtual City metaphor for the visualization of software systems and presented different techniques to improve the comprehension and scalability by providing different levels of abstractions.

We introduced the Virtual City metaphor as a basic notation for our software visualizations. The mapping we introduced is based on the following view: A virtual world is a flattened overview of the system; a city is detailed view highlighting functional group; Districts are Classes that are clustered based on their associations; and a building represents a single class as the smallest entity. The size of a city is proportional to the number of the buildings it contains. The height of the buildings maps to the size of the classes (relative to the source code). Some attributes of the 3D objects are used to represent the properties of classes. For instance, color and texture are used to indicate hotspots and to visually distinguish classes and their participation in design patterns. The city is constructed based on grid layout and grouping algorithms. This technique also scales well to size of a city.

Program analysis can provide additional insights and guidance in filtering and visualizing the information. Our grid layout technique improves the readability of 3D

visuals on screen. For example, buildings' locations are computed by taking the coupling relations among classes as parameters. Thus, their adjacency represents the coupling measurement.

We also presented a pure Java implementation and a case study on visualizing design patterns. One of the immediate results of our prototype implementation was that 3D visualization techniques can enhance and benefit the comprehension process by enhanced utilization of screen space and additional visualization effects.

However, several main challenges remain that go beyond just moving to 3D space or applying some layout or clustering techniques. Remaining key challenges are the re-creation of a mental model that closely corresponds to the mental model designers/programmers developed during the originally forward engineering process. No matter what layout, clustering or grouping algorithm one applies to identification and analysis of logical relationships among different software entities, they always will be limited by the quality of the algorithm and the lack of domain knowledge modeling. Overcoming these limitations requires additional information sources (other than source code) and domain knowledge.

With more and more applications moving into distributed and network centered environments; software visualization, analysis techniques, as well as grouping and layout approaches have to keep up with these changing requirements. Visualizing dynamic and behavioral aspects has additional challenges in the form of filtering large amount of information.

Since the JVC is the first implementation, numerous improvements and extensions are possible. In what follows, we summarize some of them.

Navigation History. It is important to "remember" the navigation history for the user so that user can trace his understanding track. This can be done by either saving the screen as a picture, or writing the scene to a VRML file.

Client - Server Visualization. Visualization result in VRML format can be transmitted over networks. Therefore, the user can submit source code at the client side. The sever side performs the automatic analysis and visualization process, and then send the result VRML file to the client side. The user can freely browse the scene via any VRML browser.

References

[AGRA 90] Agrawal H. and Horgan, J, **"Dynamic program slicing",** In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation,* SIGPLAN Notices, 25(6) , pp. 246-256, 1990.

[AGRA 93] Agrawal H., DeMillo, R., and Spafford, E., **"Debugging with dynamic slicing and backtracking",** *Software – Practice and Experience,* 23(6), pp. 589-616, 1993.

[ALPE 98] Alpert, S., Brown, K. and Woolf, B., *The Design Patterns Smalltalk Companion,* Addison-Wesley, 1998.

[ANTO 98] G. Antoniol, R. Fiutem and L. Cristoforetti, **"Design Pattern Recovery in Object-Oriented Software",** In *Proceedings of the 6th Workshop on Program Comprehension (WPC),* pages 153-160, Ischia, Italy, June, 1998.

[ATT 04] ATT Research, CIAO, http://www.research.att.com/, March 28, 2004.

[BALL 96] Ball T., Eick Stephen G., **"Software Visualization in the Large",** *IEEE Computer* 29(4): 33-43 (1996).

[BASI 95] Basili V., Briand L., Melo W., **"Measuring the Impact of Reuse on Quality and Productivity in Object-Oriented systems",** Technical

Report, University of Maryland, Department of Computer Science, CS-TR-3395, 1995.

[BASI 96] Basili V.R., Briand L.C., Melow W.L., **"A Validation of Object-Oriented Design Metrics as Quality Indicators",** *IEEE Transactions on Software Engineering*, 22 (10), 751-761, 1996.

[BASS 02] Bassil S., Keller R.K., **"Software Visualization Tools: Survey and Analysis",** *Proc. IEEE 9th Intenat.Workshop on program Comprehension* (IWPC'01), 2002, pp 7-17.

[BARR 81] Barry W. Boehm. *Software Engineering Economics*, Prentice Hall, 1981.

[BECK 94] K. Beck, R. Johnson, **"Patterns generate architectures",** In *proceedings of the 13th European Conference on Object-Oriented Programming,* Lecture Notes in Computer Science Nr. 821. 1994.

[BENE 91] M. Benedikt, **"Cyberspace: Some Proposals"**, In *Cyberspace: First Steps*, MIT Press, pp. 273-302, 1991

[BENF 94] S. Benford et al, **"Experience of using 3D graphics in database visualisation"**, Computing Department, Lancaster University, October 1994.

[BOEC 85] Boecker, H. and H. Nieper, **"Making the Invisible Visible: Tools for Exploratory Programming",** *Proceedings of the First Pan Pacific*

Computer Conference, The Australian Computer Society, Melbourne, Australia. 1985.

[BOOC 99] Booch, G., Rumbaugh J., Jacobson I. *The Unified Modeling Language User Guide,* Addison-Wesley 1999

[BROO 83] R. Brooks, **"Towards a theory of the comprehension of computer programs",** International Journal of Man-Machine Studies, 18:543–554, 1983.

[BUSH 96] Buschman, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M., *A System of Patterns,* John Wiley and Sons, New York, 1996.

[CAMP 97] Campo M, Marcos C., and Ortigosa A, **"Framework Comprehension and Design Patterns: A Reverse Engineering Approach"**, In *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering,* 1997.

[CARD 87] S.K. Card and A.H. Henderson Jr., **"A multiple virtual workspace interface to support user task switching",** *Proceedings of the CHI+GI 1987* (Toronto, April 5-7), ACM, New York, pp. 53-59, 1987.

[CFLO 04] CFLOW, http://www.gnu.org/directory/All_Packages_in_Directory/Cflow.html, March 28, 2004.

[CHAR 02] Charters S. M., Knight C., Thomas N., Munro M, **"Visualisation for informed decision making; from code to components"**, in *the Proceedings of the 14th international conference on Software engineering and knowledge engineering 2002,* Ischia, Italy 2002, pp. Pages: 765 – 772.

[CHIK 90] E. J. Chikofsky and J. H. Cross, **"Reverse Engineering and Design Recovery: A Taxonomy"**, *IEEE Software*, pages 13–17, January 1990. (pp 1, 8, 10, 13)

[COLE 94] A. Colebourne et al, **"Populated Information Terrains: supporting the cooperative browsing of on-line information"**, Research report CSCW/13/1994, Centre for Research in CSCW, Lancaster University, 1994.

[COPL 95] Coplien, James O. and Schmidt, Douglas C., *Pattern Languages of Program Design,* Addison-Wesley, 1995.

[CORT04] Cortona VRML Client, http://www.parallelgraphics.com/products/cortona/, March 28, 2004.

[COTE88] Cote, Bourque, Oligny, Rivard, **"Software Metrics: An Overview of Recent Results"**, *The Journal of Systems and Software*, 8 (1988), pp. 121-131.

[CSCO 04] CSCOPE, http://cscope.sourceforge.net/, March 28, 2004.

[CTAG 04] CTAGS, http://ctags.sourceforge.net/, March 28, 2004.

[CXRE 04] CXREF, http://www.gedanken.demon.co.uk/cxref/, March 28, 2004.

[CYGN 04] Cygnus Solutions, Source Navigator, http://www.cygnus.com/, March 28, 2004.

[DIRE 04] DirectX, http://msdn.microsoft.com/msdnmag/issues/03/07/directx90/, 18 March, 2004.

[DISC 04] Discreet, 3D Studio Max 6, http://www.discreet.com/3dsmax/, March 28 2004.

[DYNA 04] Source Dynamics, Source Insight, http://www.sourcedyn.com/index.html, March 28, 2004.

[EADE 89] P. Eades and Y. Xuemin, "**How to draw a directed graph**", In *IEEE Workshop on Visual Languages*, 13-17, (1989).

[EDEN 02] A. H. Eden, **"A Theory of Object-Oriented Design"**, *Information System Frontiers (Journal of),* Vol. 4, No. 4 (November – December 2002). Kluwer Academic Publishers, 2002.

[EICK 92] Eick, S., Steffen, J. L., and Summer, E. E., **"Seesoft - A Tool For Visualizing Line Oriented Software Statistics",** *IEEE TSE*, vol. 18, no. 11, November 1992, pp. 957-968.

[FENT 91] Fenton N., *Software Metrics: A Rigorous Approach*, Chapman and Hall, 1991.

[FERE 01] R. Ferenc, J. Gustafsson, L. Muller, J. Paakki, **"Recognizing Design Patterns in C++ programs with the integration of Columbus and Maisa"**, In *Proceedings of the 7th Symposium on Programming Languages and Software Tools (SPLST 2001)*, Szeged, Hungary, June 15-16, 2001. pp 58-70.

[FURN 86] Furnas G, **"Ggeneralized Fisheye Views"**, *Proceedings SIGCHI Human Factor in Computing*, 1986, pp 18-23.

[GAMM 93] Gamma, E., Helm, T., Johnson, R. and Vlissides, J., **"Design Patterns: Abstraction and Reuse of Object Oriented Design"**, *Proceedings of ECOOP '93*, 405-431.

[GAMM 94] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1994

[GIRA97] Jean-Francois Girard and Rainer Koschke, **"Finding components in a hierarchy of modules: a step towards architectural understanding,"** in *Proc. of the International Conference on Software Maintenance - ICSM '97.*, 1997.

[GLOB 04] Global Technologies Ltd, CODEWALKER, http://www.gtlinc.com/, March 28, 2004.

[GLUT 04] GLUT, http://www.xmission.com/~nate/glut.html, 2004.

[GOME 01] Luis M., Gomez Henriquez, *Software Visualization: an overview,* 2001.

[GOPA 91] Gopal R., **"Dynamic program slicing based on dependence relations"**, In *Proceedings of the Conference on Software Maintenance*, pp. 191-200, 1991.

[GRAM 04] GrammaTech, CodeSurfer, http://www.codesurfer.com/products/codesurfer/index.html, March 28, 2004.

[GUPT 97] Gupta R., Soffa M. and Howard J., **"Hybrid Slicing: Integrating Dynamic Information with Static Analysis"**, *ACM Transactions on Software Engineering and Methodology*, 6(4), pp 370-397, October 1997.

[HARM 98] Harman M. and Danicic S., **"A New Algorithm for Slicing Unstructured Programs"**, *Journal of Software Maintenance,* 10(6):415-441, Nov/December 1998.

[HARM 01] 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.

[HEND 86] D.A. Henderson and S.K. Card, **"Rooms: The use of multiple virtual workspaces to reduce spatial contention in a window-based graphical user interface"**, *ACM Transactions on Graphics 5*, 3 July, 1986.

[HERM 00] Ivan Herman, Guy Melançon, and M. Scott Marshall, **"Graph visualization and Navigation in Information Visualization: a Survey",**

IEEE Transactions on Visualization and Computer Graphics, 6(1):24-43, 2000

[HEUZ 03] Dirk Heuzeroth, Thomas Holl, Gustav Hogstrom, and Welf Lowe, **"Automatic Design Pattern Detection",** *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, 2003

[HORW 90] Horwitz S., Reps, T., and Binkley, D., **"Interprocedural slicing using dependence graphs"**, *ACM Transactions on Progr. Languages and Systems,* 12(1), pp. 26-61, 1990.

[HORW 92] Horwitz ,S. and Reps, T., **"The use of program dependence graphs in software engineering",** In *Proceedings of the 14th Int. Conference on Software Engineering,* pp. 392-411, Melbourne,Australia, 1992.

[HITZ 96] Hitz M., Montazeri B., **"Chidamber & Kemerer's Metrics Suite: A Measurement Theory Perspective"**, *IEEE Trans. on Software Engineering,* 22 (4), 276-270, 1996.

[IMAG 04] Imagix 4D, http://www.imagix.com/products/products.html, 18 March 2004.

[JAVA 04] Java3D, http://java.sun.com/products/java-media/3D/, 2004

[JOSE 99] Joseph B. Wyatt, "**Software visualization and Program understanding"**, University of Pittsburgh, 1999

[KAMK 95] Kamkar M and Krajina P, **"Dynamic slicing of distributed programs"**, In *International Conference on Software Maintenance*, pages 222--229, Oct. 1995.

[KNIG 99] C. Knight and M. Munro, **"Comprehension with[in] Virtual Environment Visualizations"**, *Proceedings of the IEEE 7th International Workshop on Program Comprehension*, pp4-11, May 5-7, 1999.

[KORE 97] Korel, B., **"Computation of dynamic slices for unstructured programs"**, *IEEE Transactions on Software Engineering*, 23(1), pp. 17-34, 1997.

[KORE 88] Korel, B., and Laski, J., **"Dynamic program slicing"**, In. Process. Letters, 29(3), pp. 155-163, Oct. 1988.

[KOSC 00] R. Koschke, **"Atomic Architectural Component Recovery for Program Understanding and Evolution"**, PhD thesis, Institute for Computer Science, University of Stuttgart, 2000.

[KOSC 03] Rainer Koschke, **"Software Visualization in Software Maintenance, Reverse Engineering, and Reengineering: A Research Survey"**, *Journal on Software Maintenance and Evolution*, John Wiley & Sons, Ltd., Vol. 15, No. 2, March/April 2003, pages 87-109.

[KRAM 96] C. Kramer and L. Prechelt, **"Design recovery by automated search for structural design patterns in object oriented software"**, in *Third*

Working Conference on Reverse Engineering, Amsterdam, The Netherlands, March 1996. pp. 208-21.

[KREU 99] Kreuseler, M. and Schuman, H., **"Information visualization using a new Focus + Context Technique in combination with dynamic clustering of information space"**, *Proc. of the ACM Workshop on New Paradigms in Information Visualization and Manipulation*, Kansas city, 1999, pp. 1-5.

[LAMP 95] 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.

[LARS 96] Larsen L. D. and Harrold M.J., **"Slicing Object oriented software"**, In *Proceeding of the 18th International conference on Software engineering,* March, 1996.

[LIB01] Li B., **"A Hierarchical Slice-Based Framework for Object-Oriented Coupling Measurement"**, *TUCS Technical Report* No 415. Turku Centre for Computer Science, Turku, Finland, July 2001.

[LITT 86] D. Littman, J. Pinto, S. Letovsky, and E. Soloway, **"Mental models and software maintenance"**, In *Empirical Studies of Programmers,* pages 80–98. Ablex Publishing Corporation, 1986.

[LIW93] Li W. and Henry S., **"Object-oriented metrics that predict maintainability"**, *Journal of systems and software*, 23(2), pp. 111-122, 1993.

[LETO 86] S. Letovsky, **"Cognitive processes in program comprehension",** In *Empirical Studies of Programmers*, pages 58–79. Ablex Publishing Corporation, 1986.

[LUDW01] Andreas Ludwig, RECODER Homepage, http://recoder.sf.net, 2001.

[LYLE 86] Lyle, J. and Weiser, M., **"Experiments on slicing-based debugging tools",** *Proceedings of the 1st Conference on Empirical Studies of Programming"*, pp. 187-197, 1986.

[LYNC 60] Kevin Lynch, *The Image of the City,* MIT Press, June 15, 1960.

[MACK 91] Mackinlay J, Robertson G, Card S, **"The Perspective Wall: Detail and Context Smoothly Integrated",** *Proceedings SIGCHI Human Factors in Computing,* 1991, pp 173-179.

[MALE 01] Maletic, J.I., Marcus, A., **"Supporting Program Comprehension Using Semantic and Structural Information"**, in *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001),* Toronto, Ontario, Canada, May 12-19, 2001, pp. 103-112

[MALE 01b] 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),* Toronto, Canada, May 12-13, 2001, pp. 26-35.

[MALE 02] Maletic, J.I., Marcus, A., Collard, M.L., **"A Task Oriented View of Software Visualization"**, in *Proceeding of the IEEE Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2002)*, Paris France, June 26, 2002, pp. 32-40.

[MALE 02] Maletic, J.I., Marcus, A., Feng, L. **"Source Viewer 3D (sv3D) - A Framework for Software Visualization"**, Formal Research Demonstration in *Proceedings of the 25th IEEE/ACM International Conference on Software Engineering (ICSE 2003),* Portland, OR, May 3-10,2003, pp. 812-813

[MANC99] S. Mancorids. B. S. Mitchell, Y. Chen, E. R. Gansner **"Bunch: A clustering tool for the recovery and maintenance of software structures",** In *Proc; IEEE Inter. Conference on Software Maintenance,* IEEE Computer Society Press, 1999, pp 50-59.

[MARL 90] Marlowe and Ryder, **"Properties of data flow frameworks. A unified model,"** *Acta Informatica*, vol. 28, pp. 121–163, 1990

[MART 94] Martin R., **"OO Design Quality Metrics - An Analysis of Dependencies"**, Position Paper, *Workshop on Pragmatic and Theoretical Directions in Object-Oriented Software Metrics,* OOPSLA'94, October 1994.

[MAVE 04] MAVERIK, http://aig.cs.man.ac.uk/maverik/, March 18, 2004.

[MAYR 95] A. von Mayrhauser and A. Vans, **"Program comprehension during softwaremaintenance and evolutionn"**, *IEEE Computer*, pages 44–55, August 1995.

[MAYR 97] A. Von Mayrhauser and A. Vans, **"Program Understanding Processes During Corrective Maintenance of Large Scale Software"**, *Procs. International Conference on Software Maintenance '97,* Sept. 1997, Bari, Italy, pp 12-30

[MAYR 98] 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, Ischia, Italy, June 1998. pp. 164-172

[MENG03] W. J. Meng, **"Slicing-Based Coupling Metrics"**, Master thesis, Concordia University, 2003.

[MCCO87] McCormick, B.H., T.A. DeFanti, M.D. Brown (ed), **"Visualization in Scientific Computing"**, *Computer Graphics* Vol. 21, No. 6, November 1987

[MICH01] Michaud J., Storey M.-A.D. and Muller H.A., **"Programs, Integrating Information Sources for Visualizing Java"**, *Proc. of the Inter. Conference of Software Maintenance (ICSM'2002),* Italy, 2001.

[MICR 04] Direct3D, http://www.microsoft.com/windows/directx/, 2004.

[MESA 04] Mesa, http://www.mesa3d.org/, 18 March 19, 2004.

[MORA 04] Adam Moravanszky, Source Browser, http://swix.ch/clan/admiral/TheSource/ProBrowser.htm, March 28, 2004.

[MULL 94] H. A. Müller, K. Wong, and S. R. Tilley. **"Understanding software systems using reverse engineering technology."** *The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences Proceedings* (ACFAS 1994).

[NIEL 98] Nielsen, J., **"2D is Better Than 3D"**, AlertBox, http://useit.com/alerbox/981115.html, 1998.

[NIEL 99] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis,* Springer, ISBN 3-540-65410-0, 1999.

[OPEN 04] OpenGL, http://www.opengl.org, 2004.

[PANA03] Thomas Panas, Rebecca Berrigan and John Grundy, **"A 3D Metaphor for Software Production Visualization",** *7th International Conference on Information Visualization (IV03),* London, England, July 2003.

[PARN 94] D. L. Parnas, **"Software Aging"**, In *Proceedings of International Conference on Software Engineering*, 1994. (pp 1, 7, 9)

[REKI93] J. Rekimoto and M. Green, **"The Information Cube: Using Transparencty in 3D Information Visualization",** *Proceedings of the*

Third Annual Workshop on Information Technologies & Systems (WITS'93), pp. 125-132, 1993

[PENN 87] N. Pennington, "**Stimulus structures and mental representations in expert comprehension of computer programs**", *Cognitive Psychology*, 19:295–341, 1987.

[PREE 94] Pree, Wolfgang, *Design Patterns for Object Oriented Software Development,* Addison-Wesley, 1994.

[PRIC 93] B.A. Price, R.M. Baeker and I.S. Small, **"A Principled Taxonomy of Software Visualisation",** *Journal of Visual Languages and Computing*, No. 4, pp. 211-266, 1993

[RILL 01] Rilling J., **"Maximizing Functional Cohesion of Comprehension Environments by Integrating User and Task Knowledge"**, *8th IEEE Working Conference on Reverse Engineering (WCRE 2001),* Stuttgart, Germany, October 2001, pp. 157-165.

[RILL 01b] Rilling J., Seffah A., "**Enhancing the Usability and Learnability of Software Visualization Techniques through Task Wizards and Software Agents",** *Proc. of Intern. Conference on Imaging Science, Systems, and Technology (CISST'2001),* Las Vegas, June 2001.

[RILL 01c] Rilling J., Karanth B. **"A Hybrid Program Slicing Framework",** *IEEE International Workshop on Source Code Analysis and Manipulation SCAM 2001*, Florence, Italy, November 2001.

[RILL 02] Rilling J, Seffah A., Bouthier C., "**The CONCEPT Project - Applying Source Code Analysis to Reduce Information Complexity of Static and Dynamic Visualization Techniques",** *1st International Workshop on Visualizing Software for Understanding and Analysis*, June 26 - 26, 2002 ,Paris, France , pp.90-100

[RILL 02b] J. Rilling and S.P. Mudur, "**On the use of metaballs to visually map source code structures and analysis results onto 3D space"**, In *Ninth Working Conference on Reverse Engineering (WCRE'02)*. IEEE, October 2002.

[RILL 03] Rilling J., Wang Q and Mudur S.P., **"MetaViz - Issues in Software Visualization Beyond 3D"**, Position paper *VISSOFT 2003*, Amsterdam, Netherlands, 2003, pp. 87-92.

[ROBE93] George G. Robertson, Stuart K. Card, and Jock D. Mackinlay,'Information **"Visualization Using 3D Interactive Animation",** *CACM,* Vol. 36, No. 4, April 1993.

[ROMA 92] 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.

[RUGA 96] S. Rugaber, *Program Understanding,* In A. Kent and J. G. Williams (Eds.), Encyclopedia of Computer Science and Technology, pp. 341-368, 1996.

[RUSS 00] C. Russo dos Santos, P. Gros, P. Abel, D. Loisel, N. Trichaud, and J.P. Paris, "**Metaphor-aware 3d navigation**", In *IEEE Symposium on Information Visualization*, pages 155–65. Los Alamitos, CA, USA, IEEE Comput. Soc., 2000.

[SAKA 92] M. Sakar and M.H. Brown, **"Graphical fisheye views of graphs",** In *proceedings of the ACM CHI '92*, pp. 83 - 91, May 3-7, 1992

[SCIE 04] Scientific Toolworks, Inc, Understand for C++, http://www.scitools.com/ucpp.html, March 28, 2004.

[SHNE 80] B. Shneiderman, *Software Psychology: Human Factors in Computer and Information Systems,* Winthrop Publishers, Inc., 1980.

[SHNE 92] Shneiderman, B., "**Tree visualization with tree-maps: A 2-dimensional space filling approach",** *ACM Transactions on Graphics* 11, 1 (January 1992), pp.92-99.

[SHNE 94] Shneiderman, B., "**Dynamic queries for visual information seeking**", *IEEE Software* 11, 6 (1994), 70-77. Reprinted in Card, S., Mackinlay, J, and Shneiderman, B. (Editors), *Readings in Information Visualization: Using Vision to Think*, Morgan Kaufmann Publishers, San Francisco, CA (1999), 236-243.

[SILI 90] Silicon Graphics Inc., **"Graphics Library Programming Guide"**, Document Version 2.0. Silicon Graphics, Inc., Mountain View, California, May 1990.

[SNIF04] Wind River SNiFF+, http://www.windriver.com/products/sniff_plus/, 18 March 2004.

[SOLO 84] E. Soloway and K. Ehrlich, **"Empirical studies of programming knowledge"**, *IEEE Transactions on Software Engineering,* SE-10(5):595–609, September, 1984.

[SOUR 04] "Survey of Source Code Comprehension Tools", http://grok2.tripod.com/code_comprehension.html, March 22, 2004.

[STAS92] J.T. Stasko, **"Three-Dimensional Computation Visualisation"**, GVU Center, College of Computing, Georgia Institute of Technology, Technical Report GIT-GVU-92-20, 1992

[STOR 97] M.-A.D. Storey, K. Wong, F.D. Fracchia and H. A. Müller, **"On Integrating Visualization Techniques for Effective Software Exploration"**, *Proceedings of IEEE Symposium on Information Visualization (InfoVis'97),* Phoenix, Arizona, U.S.A., pages 38-45, October 20-21, 1997.

[STOR 97b] M.-A. D. Storey, K. Wong, and H. A. M¨uller, **"How Do Program Understanding Tools Affect How Programmers Understand Programs?"**, In *Proceedings Fourth Working Conference on Reverse Engineering,* pages 12–21. IEEE Computer Society, 1997.

[STOR 99] M.-A. D. Storey, F. D. Fracchia, and H. A. M¨uller, **"Cognitive Design Elements to Support the Construction of a Mental Model during**

Software Exploration", *Journal of Software Systems,* vol. 44, pages 171–185, 1999. (pp 1, 8, 22, 55, 118, 123)

[STOR 01] C. B. M.-A. D. Storey and J. Michaud, **"SHriMP Views: An Interactive and Customizable Environment for Software Exploration"**, In *Proceedings of International Workshop on Program Comprehension* (IWPC '2001), 2001.

[TIPF 95] Tip F., **"A survey of program slicing techniques"**, *Journal of Programming Languages,* 3(3), pp. 121-189, 9/1995.

[TOGE01] [13] Together, TogetherSoft, http://http://oi.com, 2001.

[TROL 04] TrollTech, Qt, http://www.trolltech.com/, March 18, 2004.

[TZER01] V. Tzerpos, R.C Holt. "**ACDC: An algorithm for comprehension-driven clustering**", *Int. Working Conference on Reverse Engineering*, 2001

[VRML 04] VRML97, http://www.web3d.org/x3d/spec/vrml/ISO_IEC_14772-All/, March 19, 2004.

[WAIT 84] 12] W. M. Waite and G. Goos, *Compiler Construction*, Springer, New York, 1984.

[WALK 93] G.R. Walker, P.A. Rea, S. Whalley, M. Hinds and N.J. Kings, "**Visualisation of telecommunications network data**", *BT Technology Journal*, Vol. 11, No. 4, October 1993, pp. 54 - 63.

[WALK 95] G. Walker, **"Challenges in Information Visualisation"**, *British Telecommunications Engineering Journal*, Vol. 14, pp. 17-25, April 1995.

[WALK 98] Walker, R.J., Murphy, G.C., Freeman-Benson, B., Wright, D., Swanson, D., Isaak, J., **"Visualizing Dynamic Software System Information through High-level Models"**, *Proceedings of OOPSLA'98,* SIGPLAN Notices 33(10), October 1998, pp. 271-283

[WANG03] J. Q. Wang, **"Metaviz – Issues in Software Visualizing Beyond 3D"**, Master thesis, Concordia University, 2003.

[WEIS 84] M. Weiser. **"Program slicing"**, *IEEE Transactions on Software Engineering,* 10(4), July 1984.

[WELF 02] W. Löwe, M. Ericsson, J. Lundberg, Th. Panas, **"Software Comprehension - Integrating Program Analysis and Software Visualization"**, *Software Engineering Research and Practice* (SERPS), 2002.

[WELF 03] Welf Löwe, Morgan Ericsson, Jonas Lundberg, Thomas Panas and Niklas Pettersson, "**VizzAnalyzer - A Software Comprehension Framework**", *3rd Conference on Software Engineering Research and Practise in Sweden,* pages 127-136. Lund University, Sweden, October 2003.

[WEST 04] Western Wares, CC-RIDER, http://www.westernwares.com/, March 28, 2004.

[WU92] S. Wu and U. Manber. **"Agrep - a fast approximate pattern matching tool",** In *Usenix Winter 92 Technical Conference*, pages 153–162, January 1992.

[XIAN03] X. H. Xian, "**2D & 3D UML-Based Software Visualization for Object-Oriented Programs**", Master thesis, Concordia University, 2003.

[YOUN 96] Peter Young, "**Software Visualization**", Visualization Research Group Centre for Software Maintenance, University of Durham, 1996. http://vrg.dur.ac.uk/misc/PeterYoung/pages/work/documents/index.html

[ZHAN03] Y. G. Zhang, "**Automatic Design pattern Recovery**", Master thesis, Concordia University, 2003.

[ZHAO 98] Zhao J., "**Dynamic Slicing of Object-Oriented Programs"**, Technical-Report SE-98-119, Information Processing Society of Japan, May 1998, pp.17-23.