2D & 3D UML-Based Software Visualization for Object-Oriented Programs

Xiaohua Xian

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

August 2003

Your file Votre référence
ISBN: 0-612-83923-0
Our file Notre référence
ISBN: 0-612-83923-0

ABSTRACT

2D & 3D UML-Based Software Visualization for Object-Oriented Programs

Xiaohua Xian

UML (Unified Modeling Language) is a successful example of two-dimensional software visualization that is widely used in both academic and enterprise environments for object-oriented software development. The presented work (*UML3D*), which is included in the *CONCEPT* (Comprehension Of Net-CEntered Programs and Techniques) framework, applies 3D visualization techniques to UML to take advantages of 3D space and the additional features that can be applied in the 3D space. The *UML3D* project also 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. We also discuss the use of source code analysis like program slicing and coupling to improve the scalability, usability and navigability of the visual representations. An initial usability study of *UML3D* based on the SUMI (Software Usability Measurement Inventory) questionnaire was performed to study the ease of use and to identify future research directions.

ACKNOWLEDGEMENTS

I would like to thank sincerely my supervisor Dr. Juergen Rilling for his encouragement, for his guidance and support, which have made the completion of my thesis possible.

Also, I would like to thank my husband and my parents for their encouragement and support. I would like to dedicate this work to them with great love and gratefulness.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

1.1 Review

Software maintenance is an important phase in software engineering life cycle. After the first delivery of the software, there are many tasks such as error correction, environment adaptation, software evolution, etc, that need to be achieved by software maintenance. It accompanies software until the end of the use of the software. According to the statistics, over 70% of the total expenditure for software is spent on software maintenance. While in software maintenance, about 60-70% of the total effort is directed towards the comprehension of software [3]. Program comprehension is crucial to software maintenance. Effective and accurate program comprehension can save money and time for software maintainers to get into the software and achieve the tasks. There are many CASE (Computer Aided Software Engineering) tools provided to aid software engineers to understand the programs under study, e.g. software visualization tools.

Software visualization plays a significant role in program comprehension. There is an old saying: “A picture is better than 1000 words.” Software visualization uses the graphical or textural representations to make it easier for users to understand the programs. The goal of software visualization is to reduce the complexity of the programs under consideration in order to gain insight and understanding. It helps to reduce the time and energy for software maintainers to form the mental model by representing the architecture of a given software system and the relationships among the entities. Research in software visualization has flourished in the past decade. A large number of software visualization tools, techniques, and methods were proposed to deal with various problems.

1.2 Thesis Contribution

In this thesis, a new software visualization tool, *UML3D*, is proposed and included in *CONCEPT* (Comprehension Of Net-CEntered Programs and Techniques) framework.

UML (Unified Modeling Language), which is widely used in both academic and enterprise environments for software development, is a successful example of two-dimensional software visualization. It has class diagrams, sequence diagrams, collaboration diagrams, and so on.

However, it has drawbacks. First of all, it only has 2D versions of the diagrams, which don't make full use of the space. With the growing scale of large software systems and the study of cognitive issues of program comprehension, three-dimensional techniques are needed in software visualization tools to reduce the cognition overload as well as the limitation of the space usage. Secondly, it has no self-organizing layout algorithms and therefore, the representations for very large systems are a mess. Thirdly, it doesn't support intuitive navigation and interactions, which are useful for making the representations more comprehensible. Fourthly, the details of source code are lost when the diagrams are formed. The users cannot relate the software models to their corresponding source code. Lastly, the dynamic diagrams in UML – sequence diagram – cannot show the runtime behaviors of systems step by step.

In this research, we address some of the shortcomings of UML by applying three-dimensional techniques and integrating layout algorithm to it. There are four types of diagrams presented in this project. They are, all in 3D and 2D, system diagrams, class diagrams, sequence diagrams, and collaboration diagrams. System diagrams and class diagrams illustrate software in a static way, while sequence diagrams and collaboration diagrams prescribe the dynamic behavior of the system step by step. The former is based on source code while the latter is obtained from execution trace. Moreover, *UML3D* addressed some other shortcomings of UML by providing intuitive navigation and interactions with the diagrams, utilizing the techniques of filtering and clustering for information hiding and graph simplification to improve the scalability, usability and navigability. It also supports program comprehension at different levels of abstraction and allows for context switching among these different views. In addition, source code analysis, like programming slicing and coupling, which helps the maintainers to find the routines to change and reduce the entities to be presented, is also combined in the visualization. Finally, an initial usability study based on the questionnaires in SUMI (Software Usability Measurement Inventory) is performed for the evaluation of the presented work.

1.3 Organization of Thesis

The thesis is organized as follows.

In chapter 2, the overview of software maintenance, program comprehension, and software visualization is presented. UML and its diagrams are illustrated as well.

In chapter 3, a survey of 2D software visualization and 3D software visualization is given. Their comparisons are presented.

In chapter 4, the motivations of the presented work as well as the design and implementation of this project are enclosed. In chapter 5, usability study of *UML3D* based on the questionnaires of SUMI and evaluation criteria for software visualization tools is offered. The analysis result of the collected data is also given. Chapter 6 presents the conclusions and includes the discussion about future research.

Chapter 2 Background

Software maintenance plays a significant role in software engineering and the key to software maintenance is program comprehension. Software visualization techniques and their applications in reverse engineering tools are proposed to support software maintenance and program comprehension tasks [2]. In this chapter, after the introduction of software maintenance and program comprehension, the review of software visualization will be presented. The graph-based modeling language, UML, and several reverse engineering tools for visualizing software will be illustrated at the end of this section.

2.1 Software change and maintenance

Software requirements or the environment may change during the software process. In order to make the software keep up with the changes, software maintenance is employed. Software change is unavoidable. As a result, software maintenance is inevitable.

- *Legacy systems*

  Legacy systems are “older software systems that remain vital to an organization” [3]. There are risks in scrapping a legacy system and replacing it with a new system. Legacy systems are difficult and expensive to understand and maintain because the program language that was used to develop the systems may be dated, the system documentation is often old, the system structure may be corrupted by previous maintenance, etc. Current systems can become legacy systems in the future. They need improvements to keep their performance [3].

- *Software change impact analysis*

  Software change may cause side effects. A side effect is "an error or other undesirable behavior that occurs as the result of a change" [34]. To avoid the side effects, software change impact analysis is employed. Software change impact analysis estimates what will be affected in software and related documentation if a proposed change is made. The situation is even worse when software changes cause ripple effects. Ripple effect is the phenomenon where small change to a system affects many other parts of it. Ripple effect analysis is the iterative process of analyzing and eliminating side effects due to changes. Program slicing, both forward and backward, will help in ripple effect analysis (REA) [4].

- *Software maintenance*

  Software maintenance is "the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment" [4]. According to the definition, there are three types of software maintenance: corrective maintenance, perfective maintenance, and adaptive maintenance. The process of software maintenance is affected by the maintainer characteristics, program characteristics, and task characteristics.

Software maintenance has challenges. Firstly, the easiest way to add a feature is to add new code. After many changes, the software is more difficult to understand and to make further changes. There are more codes to change and it is more difficult to find the routines that must be changed. Moreover, inconsistent, inaccurate and unmatchable documentation makes the situation even worse. Documentation is crucial in the development of any large, long-lived software system. The documentation includes system architecture description, program design documentation, source code listings, test plans, validation reports and a system maintenance guide. Good documentation makes programs easier to understand. Unfortunately, documentation is always missing or out of date; the situation is even worse in legacy systems [3][4].

2.2 Program Comprehension

Program comprehension plays a significant role in software maintenance and evolution. A significant proportion of the time required for maintaining, debugging, and reusing existing code is spent in understanding existing programs. Program comprehension is "a process that uses existing knowledge to acquire new knowledge" [10]. There are various sources of information that can be used to support program comprehension: source code, documentation, reverse engineering and program analysis results, user-generated information, and maintainer's knowledge [5]. Questions listed below are concerned within program comprehension tools selection and development [6]:

1. How do programmers understand programs?
2. What tools do they need to understand programs?
3. What cognitive framework of design elements is needed to guide tool design?

Program comprehension is a time consuming activity, especially when dealing with large-scale software systems [7]. E.g. Netscape Communicator project development slowed down because of the fact that "it takes a long time for a new developer to dive in and start contributing" [2]. Thus, program comprehension is the key to effective maintenance and is the most skilled and labor-intensive part in software maintenance.

Cognitive model is useful for the construction of mental model, which refers to the maintainers' mental representation of the programs formed during program comprehension.

- *Mental model*

  A mental model describes the maintainer's mental representation of the program in question [8]. During program comprehension, the software engineer will form the mental model of a software system. This model constitutes the maintainers' understanding of the software. The mental model has the microstructure of individual program statements and the macrostructure of higher level. These constructs include text structures, chunks, plans, hypotheses, beacons, and rules of discourse.

- *Cognitive model*

  A cognitive model describes "the processes and information structures used to form the mental model" [8]. It is used for the construction of mental model during program comprehension. There are three kinds of strategies: bottom-up, top-down, and opportunistic approach.

Bottom-up comprehension

Bottom-up comprehension is built from the bottom up by reading source code and then mentally chunking or grouping these statements into higher-level abstractions. Shneiderman and Mayer's cognitive framework differentiates between syntactic and semantic knowledge. Syntactic knowledge is language dependent while semantic knowledge is language independent. Pennington's model also understands a program in a bottom-up manner. It has a program model and situation model. The program model is a control-flow abstraction, which illustrates the sequence of operations in the program that built through the chunking of microstructures into macrostructures and cross-referencing, while situation model contains data-flow abstractions and functional abstractions of a software system [8].

Top-down comprehension

Brooks understand a program in a top-down way. The process starts with a hypothesis concerning the global nature of the program. The hypotheses are generated through beacon recognition. A beacon is a set of features that indicates the existence of hypothesized structures or operations. These hypotheses are verified by searching external representations for evidence. Soloway and Ehrlich use rules of discourse and beacons help to decompose goals and plans into lower-level plans [8].

Opportunistic model

Letovsky views programmers as opportunistic processors exploiting either bottom-up or top-down cues, which are required in large-scale maintenance activities. The three components in the model are the knowledge base, the mental model, and the assimilation process. The knowledge base indicates the programmer's expertise and background knowledge. The assimilation process either bottom-up or top-down describes how the mental model evolves using the programmer's knowledge base, program source code and documentation. The central activity in the process is inquiry episodes that consist of a programmer asking a question, conjecturing and answering, and then searching through the code and documentation to support or refute the answer [8].

2.3 Software Visualization

2.3.1 Visualization

Visual is from the Late Latin word "visualis", from the Latin "visus". Visualization is the transformation of data or information into pictures. More precisely, the dictionary definition of Visualization is "the power or process of forming a mental picture or vision of something not actually present to the sight". Visualization is classified as science visualization and information visualization. Science visualization has inherent forms like the automobile industry, flow-dynamics of airplanes, chemical experiments, galaxy simulations, and many more, while information visualization usually doesn't [10].

Information visualization has many areas of application. In biology and chemistry, it includes an organizational chart and taxonomies that portray the relations between species, evolutionary trees, phylogenetic trees, food chains of creatures (see Figure 1), molecular maps, genetic maps, biochemical pathways, protein functions, DNA structure, etc [11]. It is also applied in medical areas like electrocardiograms (see Figure 2), physical areas like diagrams of optics, and so on.

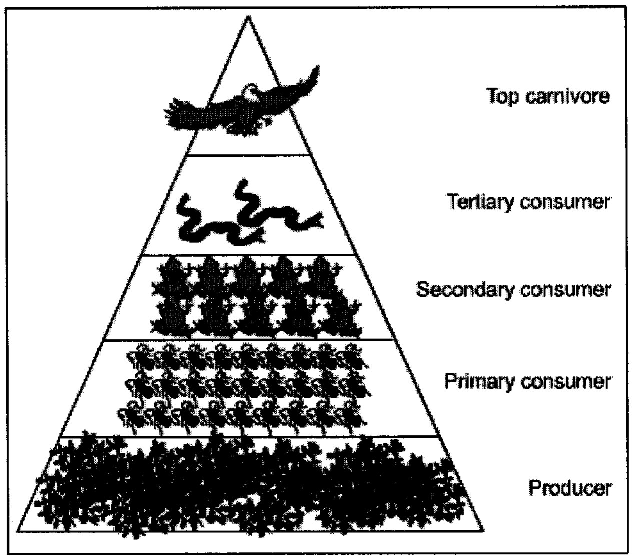


Figure 1 Food Chain: A Food Pyramid [12]

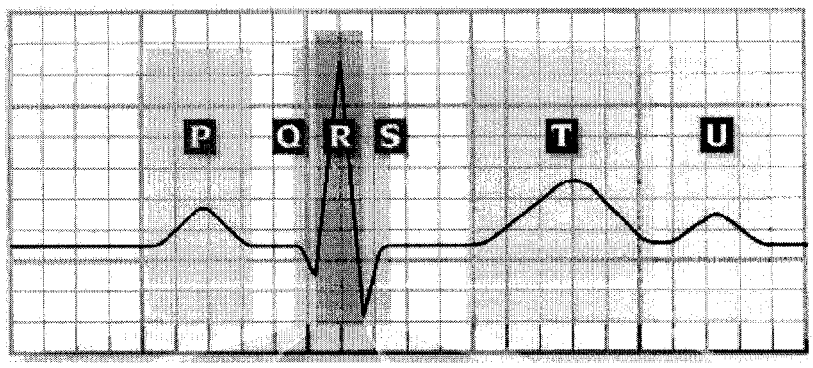


Figure 2 Electrocardiogram [13]

2.3.2 Software Visualization

"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" [7]. The goal of software visualization is to gain insight and understanding by reducing the complexity of the programs under consideration. "The purpose of visualization is insight, not virtual realities or pictures" [7]. Software visualization can be seen as a specialized subset of information visualization [10]. There are many software visualization applications, such as: web browsing, data structure, object-oriented systems' class browsers, real-time systems' state-transition diagrams, data flow diagrams, control flow diagrams, subroutine-call graphs, entity relationship diagrams, semantic networks and knowledge-representation diagrams, project management (PERT diagrams), and document management systems.

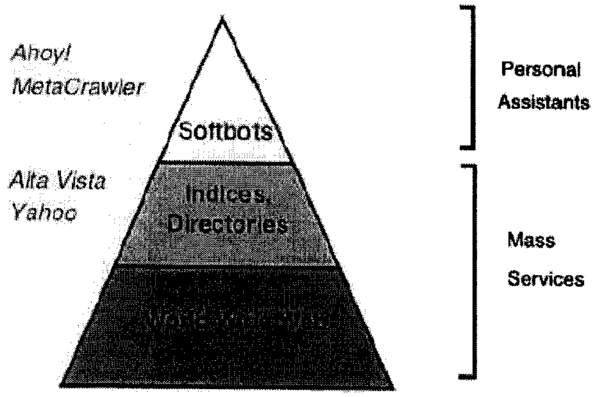


Figure 3 The Information Food Chain [15]

Some of those diagrams and presentations are converted from the traditional ones. For example, Stefan Decker et al. developed information food chains for agents that enables advanced applications on the WWW (see Figure 3) [14]. Besides that, a lot of new technologies are invented for software visualization.

The history of pictorial representations for computer programs dates back to the use of flow charts to describe control flows proposed by Goldstein and von Neumann in 1947. In 1966, Knowlton presented dynamic techniques and the visualization of data structure. Baecker, in 1968, proposed an on-line graphical debugger. Brow and Sedgewick proposed interactive algorithm animation in 1984. Research of software visualization flourished in the past decade. A large number of tools, techniques, and methods were proposed to address various problems [56].

There are five dimensions of software visualization. They are [17]:

1. Why is the visualization needed? – Tasks
2. Who will use the visualization? – Audience
3. What is the data source to represent? – Target
4. How to represent it? – Representation
5. Where to represent the visualization? – Medium

Table 1 shows the mapping of five software visualization systems along the five dimensions.

Dimension / SV System	Task	Audience	Target	Representation	Medium
SHriMP	Reverse engineering, maintenance	Expert developer	Source code, documentation, static design-level information, medium Java systems	2D graphs, interactive, drill-down	Color monitor
Tarantula	Testing, defect location	Expert developer	Source code, test suite data, error location	Line oriented representation, color, interactive, filtering, selection	Color monitor
IMSOvison	Development, reverse engineering, management	Expert developer, team manager	Source code, static design information, metrics, large OO systems	Specialized visual language, 3D color objects, spatial relationships, drill-down, interactive, abstraction mechanism	Immersive virtual environment
SeeSoft	Fault location, maintenance, reengineering	Expert developer	Source code, execution data, historical data	Line oriented representation, color, interactive, filtering, selection	Color monitor
Jinsight	Optimization	Expert developer	Program bursts, Java, dynamically collected	Color coded line oriented, text, interactive, filtering, queries	Color monitor

Table 1 Mapping of the five software visualization systems along the five dimensions [17]

Early software visualization tools like algorithm animation were aimed at supporting understanding for education purposes. Visual programming systems support domain specific programming that is especially useful in interface designing. Many of today's software visualization tools support software engineering tasks for large software systems. These tasks include development activities (e.g. programming, debugging, testing, etc.), maintenance (e.g. fault detection, re-engineering, and reverse engineering), software process management, and marketing. To summarize, software maintenance and development includes many application tasks that range from coding and debugging, to design and re-engineering. The task oriented software visualization should address different software engineering tasks by different visual representations [16][17].

Why software visualization? Software maintainers don't have enough funds, energy, and time for software maintenance. So any tools that help with maintenance should be of interest. Software visualization is important for both the writer and the reader. It is useful for debugging, testing, reusing, etc. It aids the user in constructing the mental model of the programs under study using hypertext, focus + context, shading/color/texture, dimension 2D/3D, animation, navigation, interaction, etc.

2.3.3 Terminologies

- *IA vs. AI*

  Intelligence Amplification (IA) is the use of computers to aid and enhance human intelligence. Machines are more skilled at data mining and processing while human beings are more skilled at pattern recognition [10]. Software visualization benefits the users in program comprehension due to human brain's limitations and the fact that human visual system is optimized for multi-dimensional data. Artificial Intelligence (AI) aims at trying to replace humans with machines. Software visualization is IA, not AI.

- *Program Visualization vs. Visual Programming*

  Visual programming is design-time visualization [7]. It is extremely useful in designing user interfaces. Visual programming can be seen as part of program visualization. However, when people mention program visualization, they mean visualizing existing software systems that have been delivered.

2.3.4 Taxonomy

There are several classifications for Software Visualization.

- *Classification proposed by Myers*

  Myers' classification of software visualization can be described in a two-dimensional chart (see Figure 4). Software visualization is divided into 6 groups. They are dynamic code visualization, dynamic data visualization, dynamic algorithm visualization, static code visualization, static data visualization, and static algorithm visualization [10].

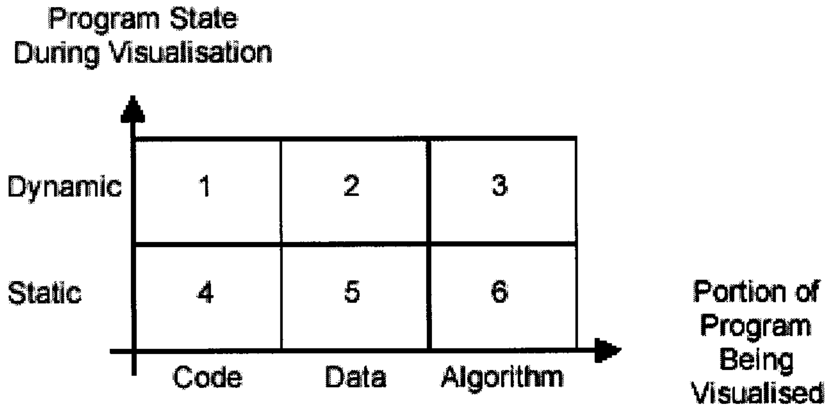

Program State During Visualisation
Dynamic
Static
1
2
3
4
5
6
Code
Data
Algorithm
Portion of Program Being Visualised

Figure 4 Visual Representation of Myers' Taxonomy [10]

- *Classification proposed by Price et al.*

  The taxonomy of software visualization given by Price et al. is a complete and detailed one and is used in the evaluation of software visualization tools for assessing and improving the design of large software visualization tools. Price et al classified software visualization into 6 subgroups (see Figure 5) [7] [18]. They are:

1. Scope: general characteristics of the visualization.
2. Content: what gets visualized?
3. Form/Method/Interaction: how is it visualized?
4. Effectiveness: how accurate and effective is the visualization?

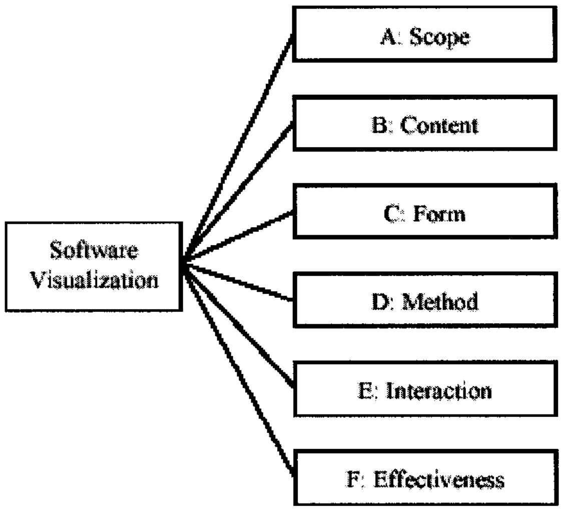


Figure 5 Price et al. Software Visualization Taxonomy [7]

2.3.5 Navigation styles

Information spaces can be databases, document repositories, the World Wide Web, or a software system. There are two navigation styles for investigating an information space: browsing and searching [19]. It follows Shneiderman's point of view for visual information seeking: "Overview first, zoom and filter, then details on demand" [37].

- *Browsing*

  Browsing is used to investigate the hierarchical composition of the software system by moving from subsystem to subsystem. A software architecture diagram is a high-level view of a software system. They are structural abstractions of the underlying software. Browsing is an unstructured exploratory strategy, with no fixed endpoint. There are a number of tools for browsing software architectures, such as Rigi, and the Software Bookshelf [19].

- *Searching*

  The visualization is generated by abstracting details from the source code to show a conceptual representation of the system. If a software maintainer wants to learn about the source code, he or she would need to access the facts underlying the abstraction. This process is called reverse abstraction. It is easier to reverse abstraction using searching. Searching is a planned activity with a specific goal, such as finding a particular fact. Searching is a powerful information seeking strategy [19].

Browsing is useful for newcomers to get familiar with the documented software system to join the maintenance team. It is also useful for other team members to understand un unfamiliar part of the system and the relations between subsystems. Searching is used for actual maintenance tasks, such as error corrections or feature addition. Browsing is used to understand the concepts, while searching is needed to reveal the facts that were used to build the concepts. Browsing uses a top-down comprehension strategy, while searching employs a bottom-up comprehension strategy [19].

2.3.6 Challenges in Software Visualization

There are several important issues concerned with software visualization.

- *Mapping metaphor*

  Metaphor creates mapping from software entities to visual representations. Good and meaningful metaphors ease the complexity of the system and quicken the process of building mental model. Shapes, colors, shading, lightning, brightness, shininess, etc. 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". However, to seek suitable mapping from the intangible software artifacts to tangible graphical elements is somehow difficult [3].

  MacKinlay defined two criteria to evaluate the mapping of data to a visual metaphor: expressiveness and effectiveness. Expressiveness refers to the capability of the metaphor to visually represent all the information we desire to visualize. The mapping from data values to visual parameters must be univocal. Effectiveness indicates the efficacy of the metaphor for representing the information. Besides, aesthetic concerns, optimization, etc. should also be regarded [17].

- *Scalability*

  The size of the graph is a major problem in visualization applications. Large software systems generate large graphs. For the large-scale systems, it is very difficult to handle the full graph at one time. Software visualization systems suffer from poor visibility, readability and usability when visualizing large, complex software system. However, there is no need for presenting the entire graph at a time. The techniques of filtering can decrease the number of entities to be presented at a time. For example, program slicing results that are based on source code analysis only show the software artifacts of interest.

Clustering techniques group elements into subsets, called clusters, based on similarity between pairs of elements. Elements have much higher similarities to the ones in the same subset than to those in different clusters. Thus, filtering and clustering are used for information hiding and graph simplification [20].

- *Layout algorithm*

  Layout is a big problem in visualization. Good layout algorithms can make the representations easier understood. There are many 2D and 3D layout techniques. However, none of them is perfect with respect to the memory and speed requirements, aesthetic issues, meaningfulness, etc. This is an ongoing research. Progress has been made and we expect to see overwhelming improvements in this field [33].

2.4 UML and Reverse Engineering Tools

UML is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software intensive system [21]. It is integrated in a number of reverse engineering tools for the purpose of facilitating program comprehension and code reusing. UML is widely used in both academic and enterprise environments for software development. However, it has drawbacks. In the presented research, we address some of the shortcomings of it and make improvements on its class diagrams, sequence diagrams, and collaboration diagrams.

2.4.1 UML Overview

Object-oriented modeling languages appeared sometime between the mid 1970s and the 1980s. Structure, which frequently referred to as their architecture, is defined clearly in UML. There are basically three different types of building blocks in UML. They are Things, Relationships, and Diagrams [30].

- *Things*

  Things are the basic entities in the model. There are four kinds of things in the UML.

  Structural things: the nouns of UML models, the mostly static parts of a model, representing elements that are either conceptual or physical. They are class, interface, collaboration, use case, active class, component, and node.

  Behavioral things: dynamic parts of UML models, the verbs of a model, representing behavior over time and space. They are interaction and state machine.

  Grouping things: the organizational parts of UML models. There is one primary kind of grouping thing, namely, packages.

  Annotational things: the explanatory parts of UML models. There is one primary kind of annotational thing, called a note.

- *Relationships*

Relationships tie things together. There are four types of relationships: dependency, association, generalization and realization.

Dependency: a semantic relationship between two things in which a change to one thing (the independent thing) may affect the semantics of the other thing (the dependent thing).

Association: a structural relationship that describes a set of links, a link being a connection among objects. Aggregation is a special kind of association, representing a structural relationship between a whole and its parts.

Generalization: a specialization/generalization relationship in which objects of the specialized element (the child) are substitutable for objects of the generalized element (the parent).

Realization: a semantic relationship between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out.

- *Diagrams*

Diagram is a graph of things and their relationships. There are 9 types of diagrams in UML. They are use case diagram, class diagram, sequence diagram, collaboration diagram, activity diagram, component diagram, deployment diagram, statement diagram, and object diagram.

Class diagrams

A class diagram shows a set of classes, interfaces, and collaborations and their relationships. It describes the static structure of a program system, consisting of a number of classes and their relationships. It illustrates meaningful concepts (classes) in a problem domain and identifies the relationships among them. Problem space is identified as and decomposed into the comprehensible concepts (classes). These concepts together clarify the vocabulary of the domain. The problem domain diagram provides a logical view of the system. Class diagrams are the most common diagram found in modeling object-oriented systems. They are important for constructing executable systems through forward and reverse engineering [30].

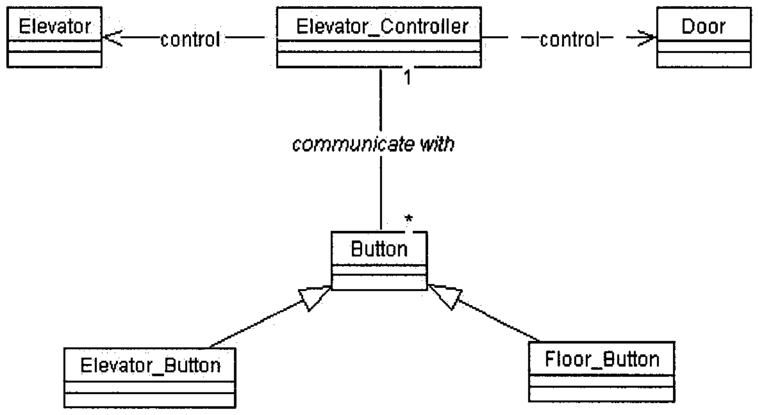


Figure 6 Example of UML Class Diagram for Elevator Button Control [9]

A class is a description of a group of objects with common properties (attributes), common behavior (operations), common relationships or other objects, and common semantics.

A package in the logical view of the model is a collection of related packages and/or classes. By grouping classes into packages, we can look at the "higher" level view of the model. In the UML, packages are represented as folders.

Sequence diagrams

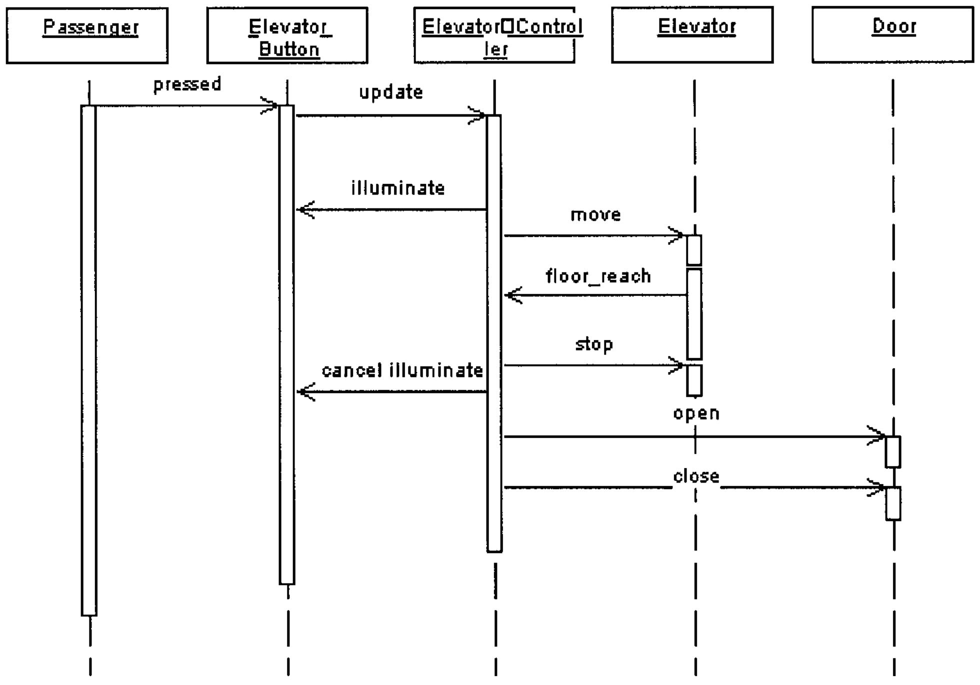


Figure 7 Example of UML Sequence Diagram for Serving Elevator Button [9]

A sequence diagram is an interaction diagram that emphasizes the time ordering of messages. In the UML, an object in a sequence diagram is drawn as a rectangle containing the name of the object underlined. An object can be named in one of three ways: the object name, the object name and its class, or just the class name (anonymous object). Each object also has its timeline represented by a dashed line below the object. Messages between objects are represented as arrows pointing from the client (sender of the message) to the supplier (receiver of the message).

Collaboration diagrams

Collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages. A collaboration diagram shows a set of objects, links among those objects, and messages sent and received by those objects. The objects are typically named or anonymous instances of classed, but may also represent instances of other things, such as collaborations, components, and nodes. You use collaboration diagrams to illustrate the dynamic view of a system.

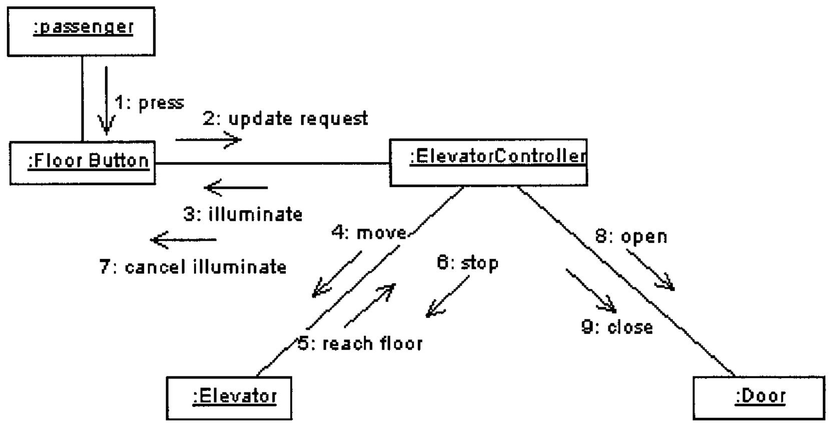


Figure 8 Example of UML Collaboration Diagram for Serving Door Button [9]

Sequence diagrams and collaboration diagrams—both of which are called interaction diagrams—are two of the five diagrams used in the UML for modeling the dynamic aspects of systems.

The presented research focuses on the diagrams illustrated above. The improvements can be applied to the other diagrams in UML as well.

2.4.2 UML in Reverse Engineering Tools

Reverse engineering is "the process of extracting and synthesizing high-level design information from source code" [22]. A reverse engineer analyzes the source code in order to identify system components and their inter-relationships and creates representations of the system in another form, usually at a higher-level of abstraction.

There are three basic activity sets in reverse engineering. They are data gathering through static analysis of the code or through dynamic analysis of the executing program, knowledge organization by organizing the raw data by creating abstractions for efficient storage and retrieval, and information exploration through navigation, analysis and presentation [3].

There are many reverse engineering tools, for example: FUJABA, Rational ... FUJABA combines UML class diagrams, UML behavior diagrams, SDM story diagrams, and design patterns to a powerful, easy to use, yet formal system design and specification language. Figure 9 is the reverse engineered UML class diagram for the famous elevator example in FUJAVA [23].

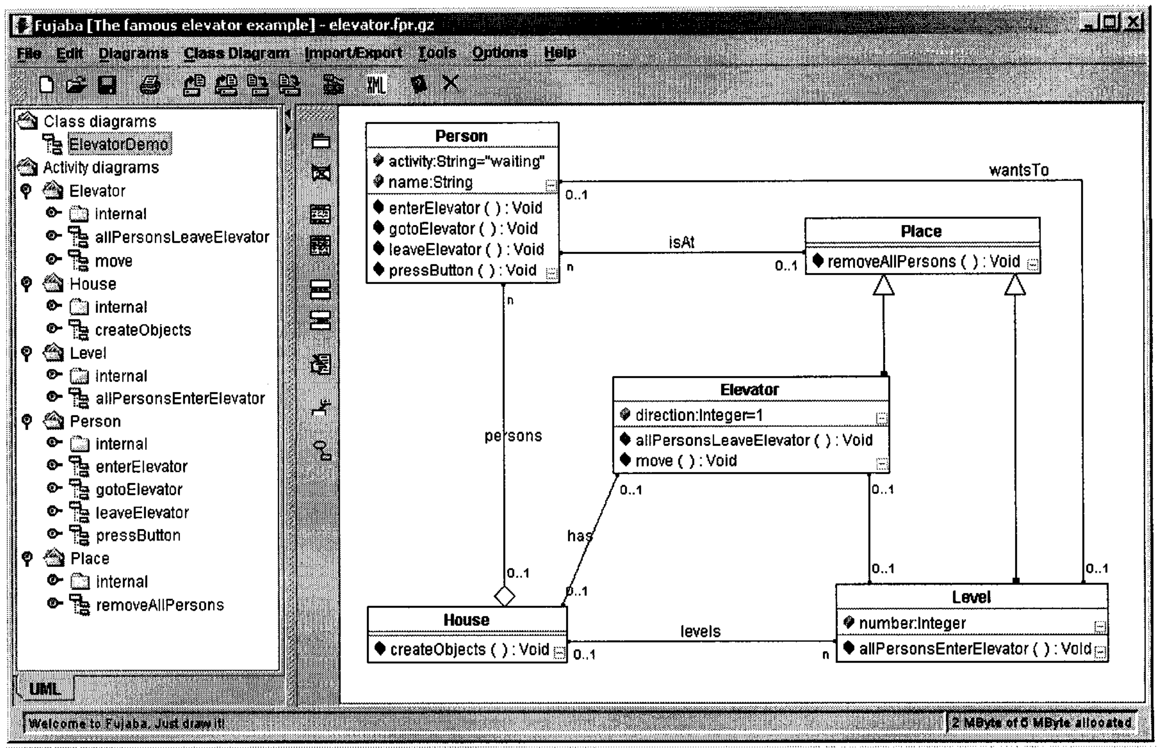


Figure 9 The famous elevator example in FUJABA [23]

The goal of reverse engineering is to extract information from the exiting software systems to better understand them [24]. The information includes the underlying features of a system, such as system structure – its components and their interrelationships, as expressed by their interface, functionality – what operations are performed on what components, dynamic behavior – system understanding about how input is transformed to output, rationale – design involves decision making between a number of alternative at each design step, and construction – modules, documentation, test suites etc [43].

Chapter 3 Software Visualization 2D versus 3D

There exist a variety of software visualization tools that are developed to facilitate the task of program comprehension, e.g. algorithm animations, visual debuggers, dynamic visualizations, pretty printers, and exploring static software structures.

There are textual representations and graphical representations that are all useful for program comprehension.

- *Textual Visualization*

  Textural visualization is the traditional source code listing which is familiar to developers as well as compilers. Program understanding is encouraged using Indentation, Hypertext, Coloring key words, Comments, etc. XML Pretty printing is an example of textural visualization [7].

- *Graphical Visualization*

  There is a saying that "A picture is better than 1000 words". Graphical visualization resembles the cognitive model of the program [3]. It is a natural way to represent relations between program elements, for example: class inheritance, call graphs, data flow, cross-reference, etc. It visualizes shapeless software information by the use of graphics. There are two-dimensional and three-dimensional graphical visualization representations.

3.1 2D Software Visualization

Two-dimensional graphical representation is a traditional and natural way for visualization. It is presented as simple xy plots. Systems supporting program visualization, animations of algorithms, computations, etc, have focused primarily on two-dimensional graphics.

There are many 2D visualization systems available. The following are some examples.

- *Treemaps*

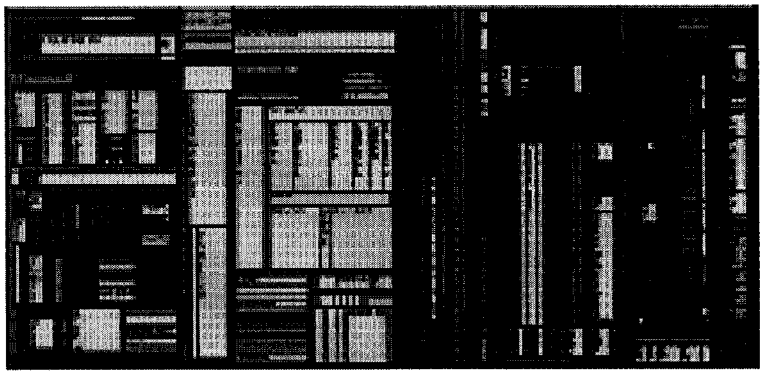


Figure 10 Treemaps [26]

The Treemaps (see Figure 10) visualization technique is a method for the visualization of hierarchically structured information [26]. It makes 100% use of the available display space, mapping the full hierarchy onto a rectangular region in a space-filling manner. This efficient use of space allows very large hierarchies to be displayed in their entirety and facilitates the presentation of semantic information.

The children of a rectangle are represented as embedded boxes laying out vertically or horizontally alternated at each level. The size of each box can be made proportional to some property, such as node size. Colors are also used to map some important features, such as the type of the files, etc. However, the rectangles are often given thin, elongated. As a result, it is difficult to compare and to select [27].

- *Seesoft*

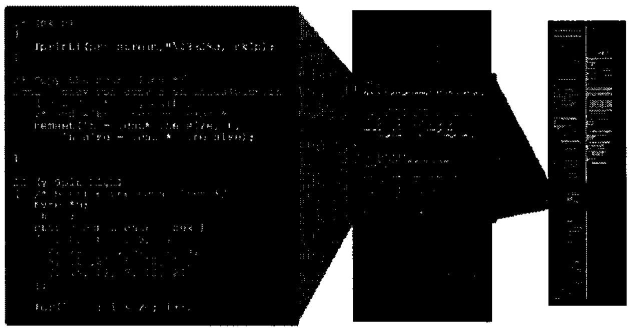


Figure 11 Seesoft [28]

Seesoft was written at AT&T Bell laboratories to be used for visualizing code for large systems. It also has been used to analyze version control data, feature location in source code, etc. Users could browse code in different levels of abstraction in Seesoft and thus obtain overview and detailed information of source code (see Figure 11). The color is used to represent the age of the code, e.g. blue represents old codes and red represents the new. Potential applications for Seesoft include project management, discovery, code tuning, and analysis of development methodology [28].

- *SHriMP view in Rigi*

SHriMP (Simple Hierarchical Multi-Perspective) was designed for visualizing and exploring software architecture. SHriMP combines hypertext metaphor with animated panning and zooming motions over the nested graph. It provides continuous orientation and contextual cues for the user that allows browsing source code while knowing the location in the software hierarchy [22].

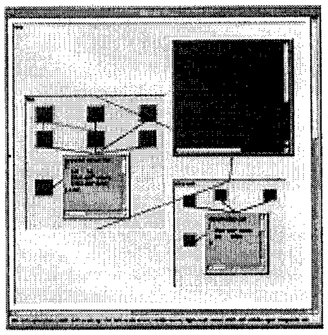
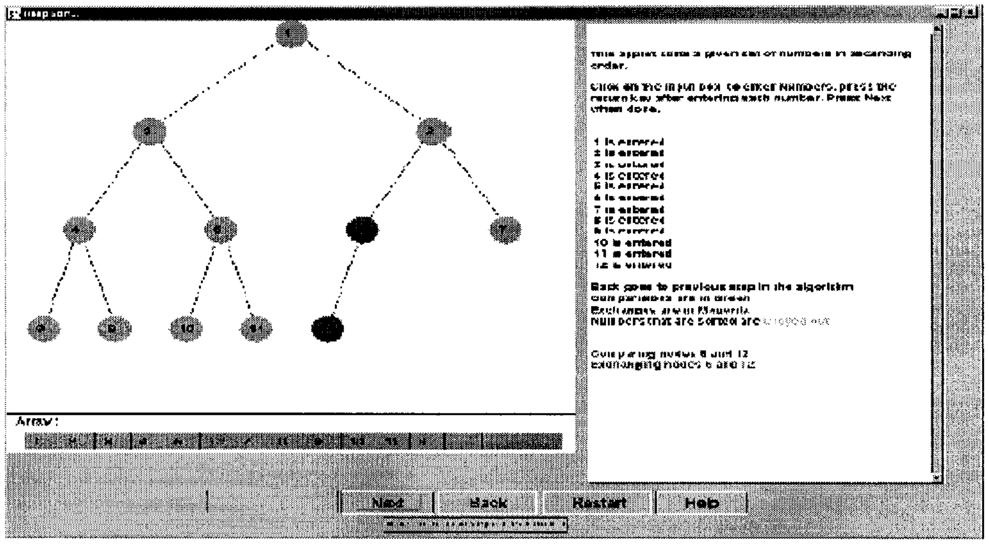


Figure 12 SHriMP view in Rigi [22]

- *The Swan System*

Swan is a data structure visualization system for computer science education. It allows users to visualize data structure and the basic execution process of a C or C++ program. Figure 13 shows views of the heapsort algorithm. The view on the bottom left is the physical array storing the values to be sorted. The view on the top left is the logical heap structure. It allows users to control the speed of algorithm execution by providing “Next” and “Back” buttons to continue or revert to steps [29].


This applet sorts a given set of numbers in ascending order.

Click on the input box to enter numbers, press the return key after entering each number. Press Next when done.

1 is entered
2 is entered
3 is entered
4 is entered
5 is entered
6 is entered
7 is entered
8 is entered
9 is entered
10 is entered
11 is entered
12 is entered

Back goes to previous step in the algorithm
Comparisons are in green
Exchanges are in [illegible]
Numbers that are sorted are [illegible]

Comparing nodes 6 and 12
exchanging nodes 6 and 12

Figure 13 Heapsort in Swan [30]

3.2 3D Software Visualization

Today's software systems are increasingly large and complex. This makes the tasks of programming, understanding, and modifying the software more and more difficult [18]. Because of the limitations of 2D representations, 3D computer graphics were introduced to software visualization.

3.2.1 Why 3D?

3D techniques are initially applied in game design. It has many advantages in contrast to 2D. Firstly, 3D makes efficient use of space compared to 2D [11]. One of the major problems addressed in graph visualization is the size of the graph. The size of a graph can make a normally good layout algorithm completely unusable. There are few systems that can deal effectively with thousands of nodes. When the scale of software increases, the visibility, usability, and discernibility of the graph visualization accordingly experience dramatic drops. Also, the high density of the layout makes the interaction, navigation, and query about particular nodes very difficult and even impossible (see figure 14). 3D has one more extra dimension that can be used to encode some knowledge compared to 2D [10]. It works better for high dimensional data than 2D views. 3D has much greater working volume and has more flexibility to represent and organize the information. The efficient use of a 3D space for visualization is proposed as a solution to overcome the limitation of available exploration space and the problem of link crossing. Thus, the use of 3D is proposed to alleviate these problems.

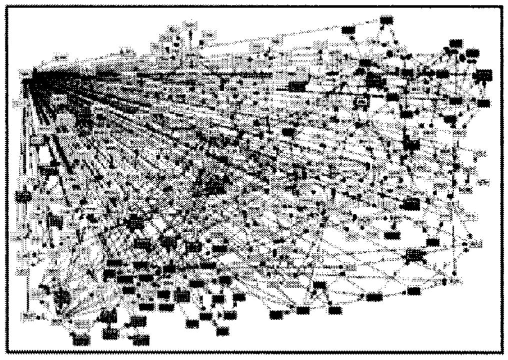


Figure 14 Call graph from a medium size system

Secondly, according to cognitive science and human perceptual system, 3D graphical representations give less cognitive strain because of familiarity and realism. "Three-dimensional displays help shift the viewing process from being a cognitive task to being a perceptual task. This transfer helps to enable humans' pattern matching skills" [7]. 3D software visualizations provide intuitive navigation and interactions. The human's perceptual, cognitive and institutive skills can be used in a 3D environment [31]. Thirdly, the error rate in identifying routes in 3D graphs is much smaller than 2D graphs

[32]. Lastly, PC infrastructure has rapidly improved over the last 3 years. It is now feasible to shift to 3D because of the current hardware and technology advances and the dramatically falling cost.

3.2.2 3D Techniques

To exploit the full range of the 3D possibilities, a lot of advanced 3D techniques are applied in 3D representations. The raw classifications of 3D techniques are given below [33].

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

  Surface Plots: It is also extended from 2D and is used to identify features such as patterns or irregularities.

  Cityscapes: It is an extension of 3D bar charts and a variation to surface plots. It aids simple comparison of results.

  Benediktine space: 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.

  Spatial arrangement of data: The techniques require that the user can interpret the properties of data items from their position and presentation within the virtual environment.

- *Dynamic Information Visualization Techniques*

Fish-eye views: Fish-eye views 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.

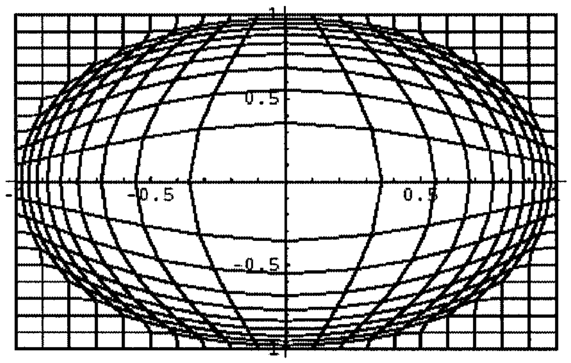


Figure 15 Fish-eye views [33]

Emotional icons: Emotional icons 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: It refers to the techniques used in automatically laying out graphs. The system arranges the graphs from unstable to stable according to efficiency, speed, accuracy and aesthetics.

- *Information representation techniques*

Perspective walls: It is used to view and navigate large linearly structured information. It 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.

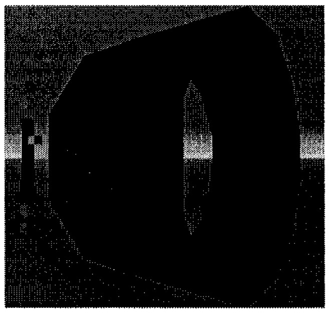


Figure 16 Perspective walls [33]

Cone trees and cam trees: Cone tree is one of the best-known 3D graph layout techniques in visualization [33]. 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.

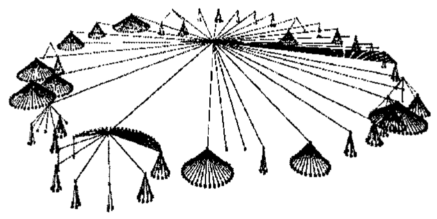


Figure 17 Cone trees and cam trees [33]

3D-Rooms: 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.

Information cubes: Information cubes 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.

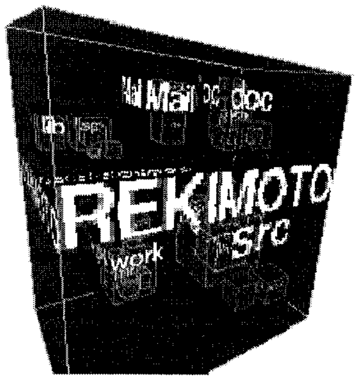


Figure 18 Information cubes [33]

3.2.3 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 that do not exist in the real world. The simplest form of virtual reality is a 3-D 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. Both representations offer the perception of depth. The goals of Virtual Reality are that of creating the illusion of submersion in a computer generated environment. Virtual reality and 3D graphical environment has become commonplace nowadays. The use of it 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 a new life to software visualization. However, VR is not a cure-all. We should not misconceive and exaggerate its power [7].

3.2.4 3D Software Visualization Systems

- *Hyperbolic tree*

  Hyperbolic browser is a focus + context technique for visualizing and manipulating large hierarchies [35]. The components in the representation diminish in size as they move outwards. The hyperbolic browser initially displays a tree with its root at the center and then smoothly transforms to bring other nodes chosen by the users into focus. The context always includes several generations of parents, siblings, and children, making it easier for the user to explore the hierarchy without getting lost.

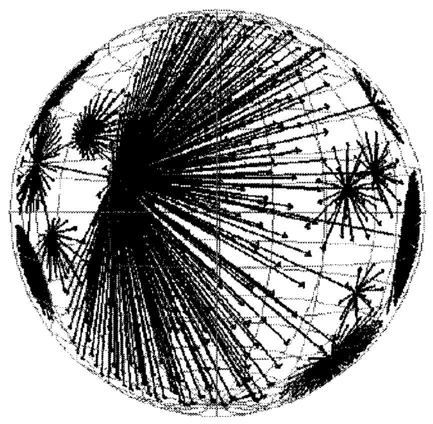


Figure 19 Hyperbolic tree [33]

- *Software World*

The software world is targeted at the visualization of Java code. It mapped 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 [2].

Figure 20 Software World [2]

- *Cone tree*

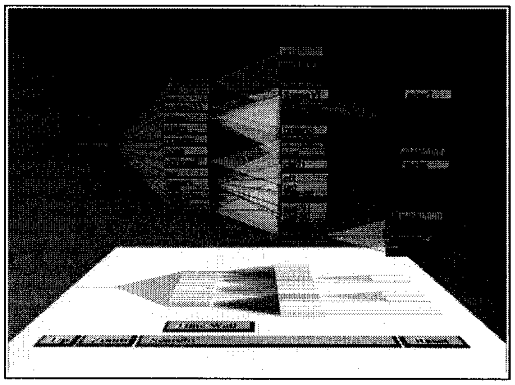


Figure 21 Cone tree [33]

The cone tree is an animated 3D Visualization of Hierarchical Information. Users can interact with the representation by moving around within the scene, rotating cone trees, zooming into packages, or fading out some details of a lower degree of interest [36].

- *SGI File System Navigator*

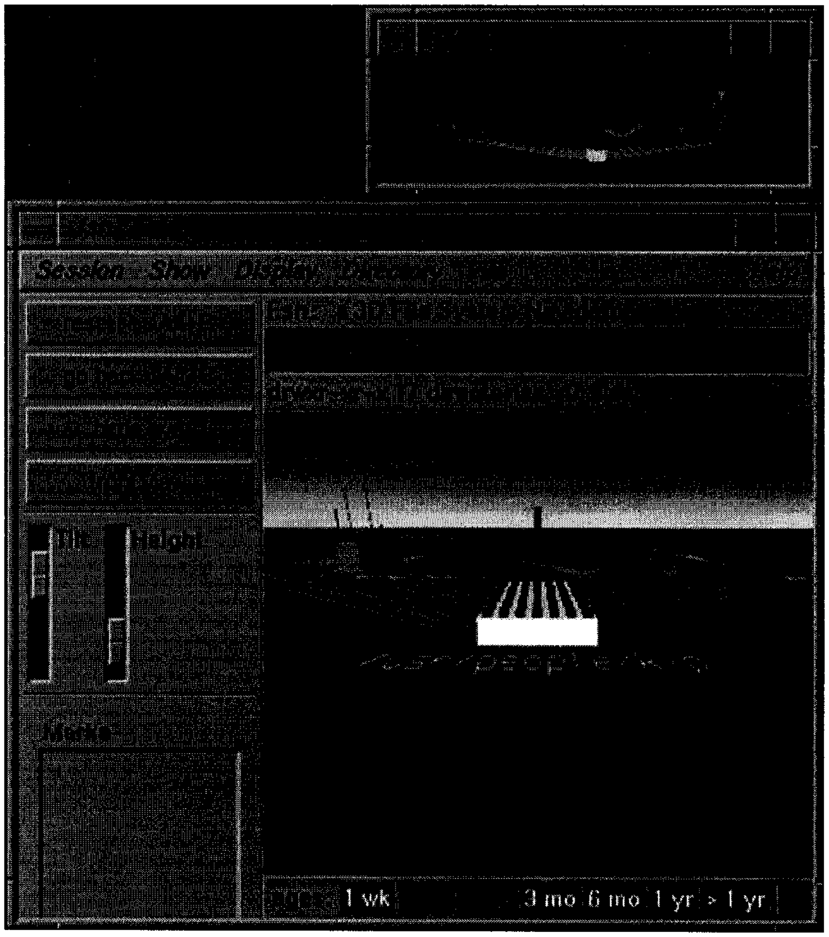


Figure 22 The SGI File System Navigator [52]

SGI file system navigator lays out the directories in a hierarchy with each directory represented by a pedestal. The height of the pedestal is proportional to the size of the files in the directory. The directories are connected by wires, on which it is possible to travel. On top of each directory are boxes representing individual files. The height of the box represents the size of the file, while the color represents the age [52].

- *MetaVis*

MetaVis (Metaball Visualization) is a new software visualization system [37]. It uses metaballs, a 3D modeling technique that 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. MetaVis is useful for showing the couplings of software artifacts.

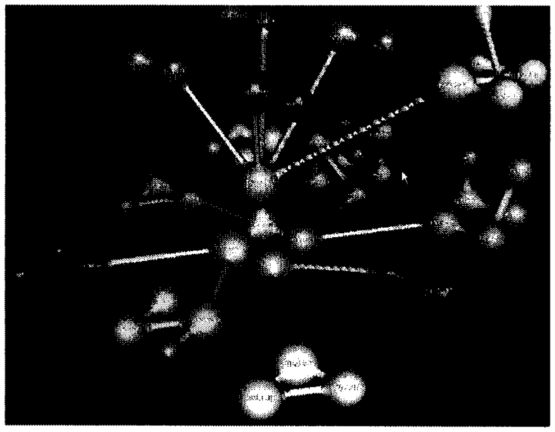


Figure 23 MetaVis [40]

3.3 Challenges of 3D versus 2D

As mentioned in the previous section 3D visualization techniques have often significant advantages over 2D visualization techniques,; however it confronts also many new challenges as well.

* *Requirements*

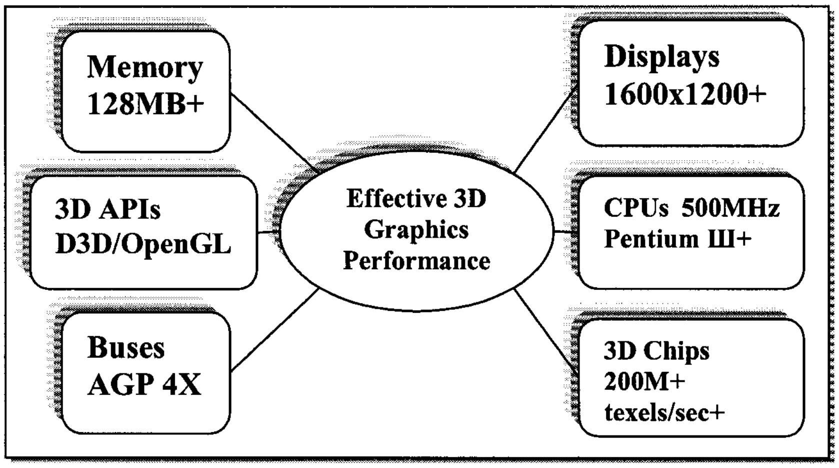


Figure 24 Requirements for 3D [38]

Three-dimensional visualization has much higher hardware requirements than two-dimensional visualization, such as memory size, CPU speed and graphic adapters, which should support natively 3D. Thus, 3D needs more hardware and software supports than 2D (see Figure 24).

* *Ease of use*

Three-dimensional visualization has more complex interfaces and interactions than two-dimensional visualization. Multiple windows are confusing and difficult to manage [33]. There is a learning curve for users to get into the use of the 3D systems.

* *Navigation Orientation*

Moving in 3D is complicated. Objects in 3D can obscure one another and it is also difficult to choose the best view in space. Changing the user's viewpoint may cause disorientation [39]. Navigating without being lost, you need to know "Where am I?" "Where is the target I am looking for?" [7] Thus, orientation cues are needed for navigation.

- *Computational complexity*

3D visualization usually has heavy demand on computation and thus makes it difficult for run-time visualization. Some proposed layout algorithms need a lot of computations. Time complexity of an algorithm indicates the need for near real-time interaction. The interactions with the graph are even more difficult if the algorithm has high time complexity. This makes the task of keeping the effective performance of the 3D world even more difficult.

- *Layout*

Layout is an important issue in visualization. Good layout can ease the complexity of the graph and can thus make the comprehension task easier. There are many layout algorithms available; however, none of them are perfect. Especially for 3D layouts, there are more issues to be taken into consideration like obscurity, orientation, etc. These make it even more challenging for achieving the goal of 3D layouts.

- *Cost*

As we mentioned above, 3D has much higher requirements than 2D. The design and implementation of 3D is not straightforward. As a result, the 3D world is more expensive and more complicated to build.

Chapter 4 Contributions

4.1 Motivations

The graph-based modeling language – UML – is a successful example for two-dimensional software representations. Its notations and diagrams are well known to software engineers. As we discussed before in chapter 3, three-dimensional software visualization has many advantages in contrast to 2D software visualization, like more efficient use of space, less cognitive strain, smaller error rate in identifying routes. Thus, the motive for applying 3D techniques to UML is obvious.

Moreover, there is no self-organizing layout algorithm in traditional 2D UML. Without good layout algorithms, the representations are becoming less readable and therefore less usable. Users cannot identify clearly the relationships among the different artifacts and therefore the comprehension task is unnecessarily complicated. In this project, an improved grid layout proposed by J. Wang [40] with the goal to minimize the edge crossings is employed for both 2D and 3D UML visualizations.

In addition, especially for very large software systems, the visualization techniques have to deal with a large amount of information. Even 3D techniques and good layout cannot resolve the problems like poor visibility, usability, discernibility, etc. According to the studies performed investigating human cognition and perceptual nature, there is no necessity for representing all the data at one time. Clustering and filtering can be employed to alleviate the situation by only presenting the information of interest while hiding other information.

Lastly, traditional UML is based only on the structural representation of the underlying system and does not require any type of source code analysis to refine the visualized information. Source code analysis can play an important role in guiding programmers during the comprehension process of larger software systems. As part of this project, we utilize program slicing to refine our presented visualization techniques to further enhance their applicability. In particular, we are using program slicing and coupling measurements to provide additional guidance to the users.

4.2 Design and Implementation

In this section, we first introduce the CONCEPT project and the system architecture for *UML3D*, and then provide design and implementation details of our 3D UML approach. Some of the major features of the system will be presented at the end the section.

4.2.1 System Architecture

The CONCEPT (Comprehension Of Net-CEntered Programs and Techniques) project addresses important software comprehension and maintenance issues such as the development of new comprehension techniques and their integration in the CONCEPT framework, etc. The project is a continuation of the previous MOOSE (Montreal Object

Oriented Slicing Environment) project [41].

The system architecture for *UML3D* is illustrated in Figure 25. The process can be divided into three phases: Source Code Parsing phase, Analysis phase, and Design Recovering phase [43].

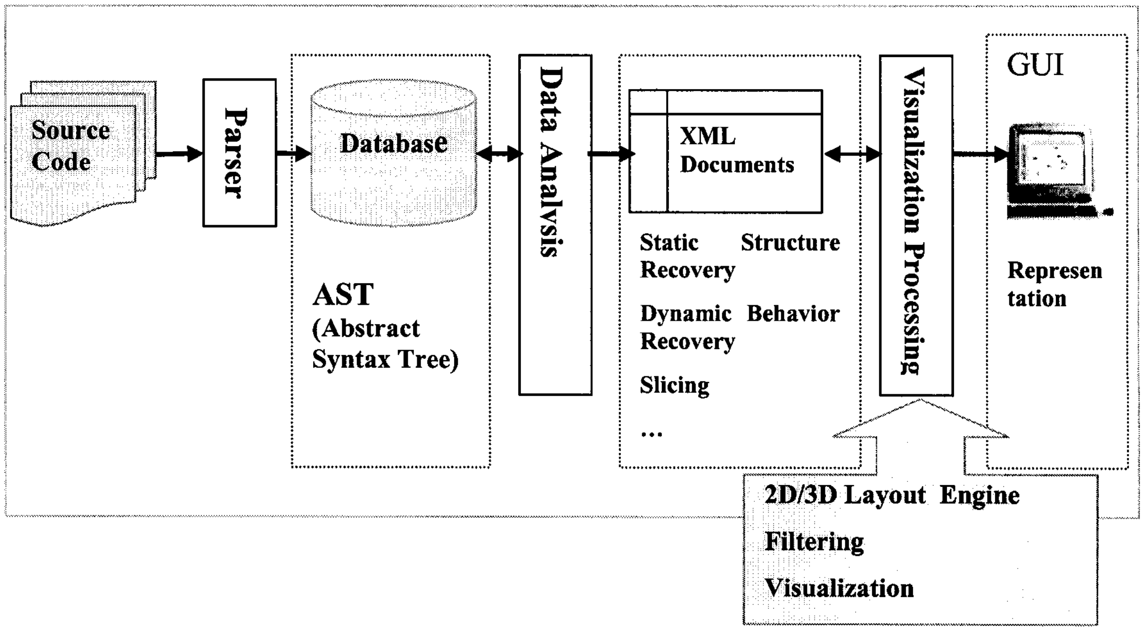


Figure 25 Architecture for *UML3D*

- *Source Code Parsing Phase*

  In this phase, an intermediate representation – AST (Abstract Syntax Tree) – is created by Javac, which can be accepted as input of the next phase. The AST is obtained by analyzing source code and extracting information from the source code and stored in PostgreSQL database. It includes all the information for the program, such as class information, variable information, etc.

- *Analysis Phase*

The component's artifacts are revealed from this phase such as static structure information recovered from AST, dynamic behavior information extracted from execution trace, slicing & coupling results obtained from analysis, etc. The data obtained by this analysis process is stored as XML files. An example of the XML file that is used to store data for the static diagrams (system diagrams and class diagrams) in *UML3D* is shown in appendix A.

The extensible markup language (XML) has been applied successfully in a lot of application domains and is found useful for data exchange in visualization [44]. The need to exchange data and metadata between incompatible Computer Aided Software Engineering (CASE) tools leads to development of standards such as the CASE Data Interchange Format (CDIF).

- *Design Recovering Phase*

In the last phase, the layout machine computes the layout data of the visualization and stores the results as XML files that will be accessed by the visualization procedure.

The design model generated in this phase offers not only functionality and system behavior, but also high-level views of the underlying software architecture. It can be used to illustrate both dynamic and static information of a software system.

4.2.2 System Design

- *Class model*

The *UML3D* system is an UML-based software visualization tool for visualizing object-oriented software programs such as JAVA. The tool includes a self-organizing layout algorithm to minimize the crossings in an existing layout. The *UML3D* tool supports views at different levels of abstractions, allowing users to switch among these abstraction levels, depending on the particular comprehension task. The abstraction levels supported by the system are on the package level, class level and statement level. All of these levels represent important information about object-oriented programs and their structures. Besides the static information, *UML3D* also provides visualization support to represent the dynamic behaviors of a software system. Within the *UML3D* environment, we utilize sequence and collaboration diagrams to illustrate the execution sequence and call relations of a program. Program slicing is applied to improve the scalability and performance for very large software systems to reduce the amount of information to be displayed and to reduce the complexity of the representations. Moreover, the *UML3D* supports both key navigation and mouse interactions. Figure 26 shows the high-level class model for *UML3D*.

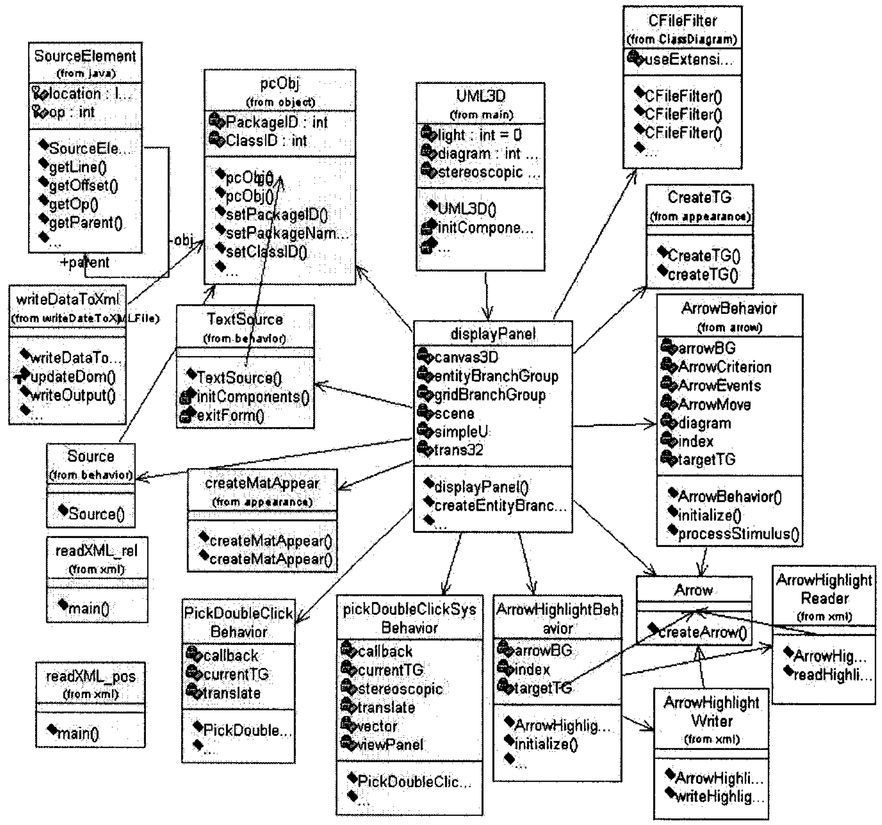


Figure 26 The Primary Class Model for *UML3D*

- *Mapping*

The goal of mapping is to find a meaningful and intuitive correspondence between the visualization techniques and the software artifacts displayed. The mapping metaphor in *UML3D* is based on traditional UML notation, however we introduce some additional extensions. We use shape for different category of data, while size is useful for quantitative data. In *UML3D*, classes and objects are mapped to boxes while packages are mapped as spheres. The text attached on the surfaces of the entities corresponds to the names of the packages, classes, or objects. Color is used to show the relationships between software systems elements. For instance, in *UML3D*'s class diagrams, classes within the same package are in the same color. The collection of materials used in *UML3D* is referenced from CUGL (Concordia University Graphics Library) provided by Prof. Grogono [42]. Table 2 shows the mapping applied in the current implementation of *UML3D*.

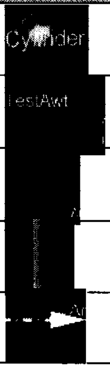
SOFTWARE ARTIFACTS	CHARACTERS OF *UML3D*	EXAMPLES
Package	**Sphere, Color**	
Class	**Box**	
Relationship	**Arrow**	
Time-Line	**Cylinder**	
Current Call	**Highlighting**	

Table 2 Mapping Metaphor for *UML3D*

- *Data format*

The *UML3D* plugin is based on a layered system architecture and is independent from the other parts of the analysis tools. The UML3D tool accepts a simple and flexible XML as an input. The output of the numerous analysis tools can be easily translated into the appropriate XML format used by *UML3D* via XSLT, which is a language for transforming XML documents into other XML documents [53]. Details of the XML file format in *UML3D* visualization pipeline can be found in appendix A.

- *Extensible design and implementation*

Compatibility and extension are good qualities of any software system. One of the major design issues for the *UML3D* was to make both its design and its implementation extensible. At the current development stage, the tool supports four types of diagrams. However, further functions and other types of diagrams can be added easily to allow for extendibility and customization of the tool, supporting future evolution of the tool. This is achieved by utilizing a layered architecture approach with most of the classes corresponding to a design based on low coupling and high cohesion.

- *Integration with MetaVis, Software city*

  The *UML3D* tool will be integrated with both MetaVis and Software city. Data can be simultaneously displayed using either of these visualization techniques. This provides the user with the ability to switch among different views and context. It is anticipated that it may lead to the development of new or the extension of existing visualization techniques by comparing and evaluating the current implementation of the visualization techniques.

4.2.3 Developing Tool

One of the major advantages of 3D visualization techniques is their ability to provide more flexibility in presenting and navigating large amount of information. The development of 3D visualization techniques benefits from the availability of libraries and tools supporting directly the 3D development (e.g. Java 3D, DirectX, etc).

- *Comparison among 3D tools*

  The following is the table of comparison among several tools such as OpenGL, VTK (Visualization Toolkit), Java3D, etc. As we can see from Table 3, Java3D has high platform flexibility, high visualization flexibility, is easy to use, has moderate learning curve, and no cost. Consequently, Java3D was chosen as the ideal tool development environment for this project.

Item	Platform Flexibility	Visualization Flexibility	Implementation	Ease of Use	Learning Curve	Cost	Other
eBizinsights	High	*Low*	Needs server/DB	Easy	Low	*$30K*	*For websites*
aiSee	High	*Low*	*Need GDL input*	Fairly easy	Fairly Low	$355 - $455	*Could not contact, not US company*
Inxight	High	Medium	J2EE Servers and Java/Windows Environment	Moderate	*Fairly High*	*$75K*	*Evaluation copies very difficult to run,*
SHriMP	*Low*	Medium	*UNIX only*, C, some pre-processing of code	Fairly easy	Fairly Low	SHriMP is free—commercial implementations will cost.	*SHriMP is a research product.*
CodeTest Suite	*Low*	Medium	Platform flexible, must purchase additional IDE	Easy	Low	Vendor did not reply	*Oriented towards code development*
xSUDS	High	*Low*	Windows and Unix,	Moderate	***High***	Vendor did not reply	*Oriented towards code development*
OpenDx	High	*Medium*	3D graphics card	*Moderate*	*Moderate*	Free	Good 3rd party modular support
Java/Java3D	High	High	3D graphics card	Easy	*Moderate*	Free	
VTK	High	High	3D graphics card	*Fairly Difficult*	*Moderate*	Free	
OpenGL	High	High	3D graphics card	***Moderate***	*Moderate*	Free	
Qt	High	High	3D graphics card	Easy	Very Low	Commercial $1500; otherwise free.	Well documented, integrates easily with OpenGL

Table 3 Making a "Tool" choice [57]

- *Java3D application scene graph*

Java3D is an extension of Java for displaying three-dimensional graphics, which is built on top of OpenGL. The programmer works with high-level constructs for creating and manipulating 3D geometric objects. The geometric objects reside in a virtual universe, which is then rendered. A Java 3D program creates instances of Java 3D objects and places them into a scene graph data structure. The scene graph (see Figure 27) is an arrangement of 3D objects in a tree structure that completely specifies the content of a virtual universe, and how it is to be rendered [46].

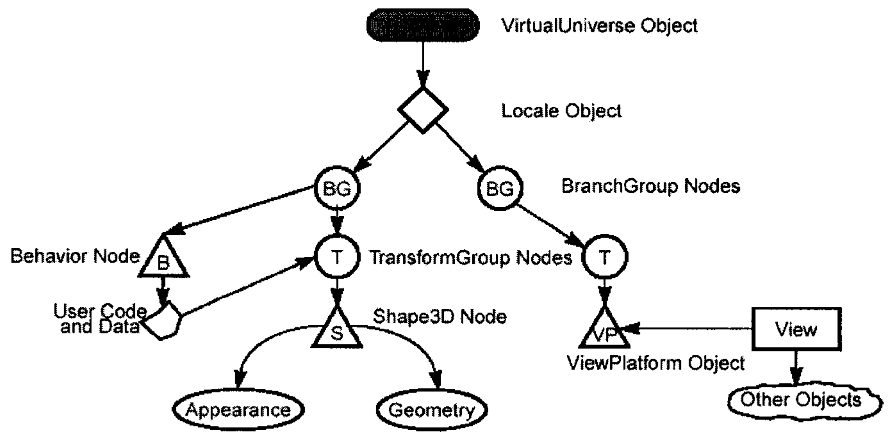


Figure 27 Java3D Application Scene Graph [46]

Below the VirtualUniverse object is a Locale object. A scene graph is formed from the trees rooted at the Locale objects. The Locale object defines the origin of its attached branch graphs. The scene graph itself starts with the BranchGroup nodes. A BranchGroup serves as the root of a subgraph, called a branch graph of the scene graph. Only BranchGroup objects can attach to Locale objects. One subgraph in this example consists of a user-extended Behavior leaf node. The Behavior node contains Java code for manipulating the transformation matrix associated with the object's geometry. The other subgraph in this BranchGroup consists of a TransformGroup node that specifies the position (relative to the Locale), orientation, and scale of the geometric objects in the virtual universe. A single child, a Shape3D leaf node, refers to two component objects: a Geometry object and an Appearance object. The Geometry object describes the geometric shape of a 3D object while the Appearance object describes the appearance of the geometry such as color, texture, material reflection characteristics, and so on [45].

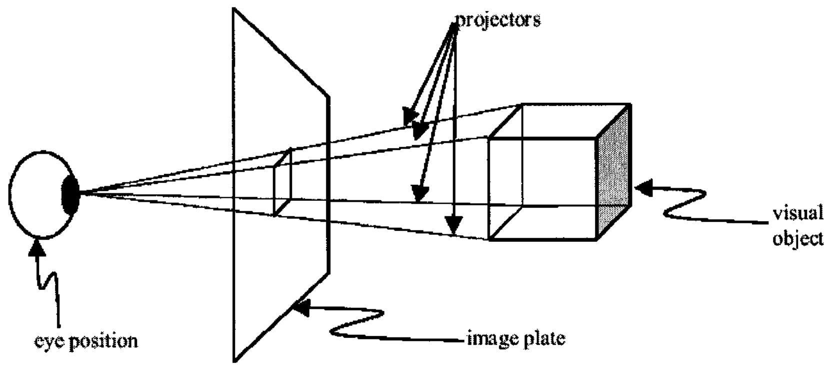


Figure 28 Image Plate and Eye Position in a Virtual Universe [47]

The TransformGroup on the right specifies the position (relative to the Locale), orientation, and scale of the ViewPlatform. This transformed ViewPlatform object defines the end user's view within the virtual universe. The ViewPlatform is referenced by a View object that specifies all of the parameters needed to render the scene from the point of view of the ViewPlatform. Also referenced by the View object are other objects that contain information, such as the drawing canvas into which Java 3D renders, the screen that contains the canvas, and information about the physical environment. Figure 28 shows the conceptual drawing of image plate and eye position in a virtual universe.

4.2.4 Implementation Issues

The *UML3D* tool has been implemented in Java3D, using the JFC/Swing components. The primary design of the *UML3D* classes is shown in Figure 26. In what follows, we describe some of the basic classes and their implementation in more detail.

- *UML3D scene graph*

In this project, the application scene graph can be drawn as below (see Figure 29).

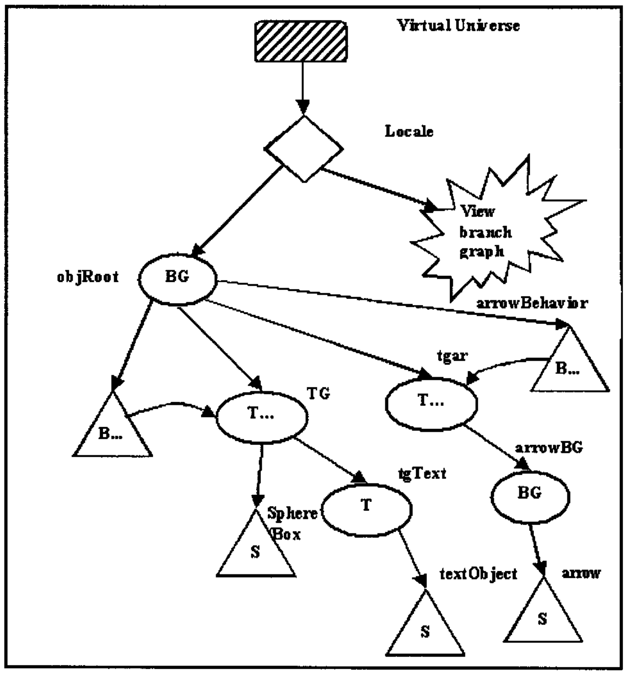


Figure 29 ***UML3D*** **Scene Graph**

In this scene graph, objRoot is the branch group, which is the root of graph. Under objRoot, there are a number of TGs. TG corresponds to the transform group that decides the position, scale, and orientation of its children nodes in the scene. The TGs values are obtained from the underlying layout algorithm within the *CONCEPT* project. The layout algorithm optimizes the layout for the minimum number of edge crossings. Under each TG, its children are added such as a sphere representing a package in a system diagram and a box portraying a class in a class diagram. The text on the surface of a graphical entity describes the name of a package or a class. The data are retrieved from XML files where the results of structural information of a software system are stored. The relationships among the software components, such as call, coupling, etc, are displayed as the arrows pointing from the client class to the server class.

- *Arrow implementation*

The Java3D library provides already many primitive drawing entities such as sphere, box, cylinder, etc. However, the library lacks some more specific and refined entities, for example the implementation of an arrow. As part of this research, the Java3D library was extended to provide the missing functionality.

Arrow Creation

The arrows are used to represent relationships between packages, classes, and objects. The arrow is composed of two components: Cone and LineArray. The line is drawn between the two points. The pattern of the line such as dash, dot, solid, etc. can be used to indicate the different relationships among the software artifacts. The arrow cap – Cone – is centered at the origin with the given radius and height. To draw a correct arrow, we should first rotate the straight upward cone to point from one point to the other. The code used to rotate the cone is followed and the variables are shown in Figure 30.

```
Vector3d v3darrow = new Vector3d(arrow3d);

Vector3d v3dcone = new Vector3d(0,1,0);

double v3dangle = v3dcone.angle(v3darrow);

Vector3d v3daxis = new Vector3d();

v3daxis.cross(v3dcone,v3darrow);

AxisAngle4d rot = new AxisAngle4d(v3daxis,v3dangle);

Transform3D tgr = new Transform3D();

tgr.setRotation(rot);
```

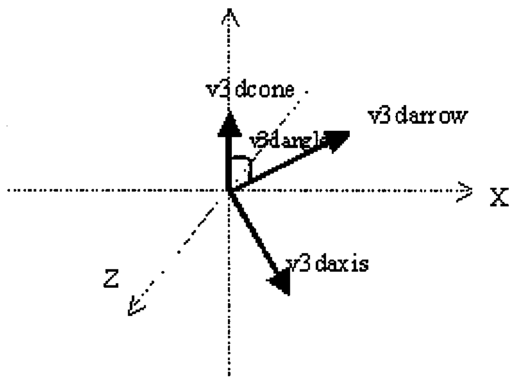


Figure 30 Rotation of the Arrow Cap

Next, the cone should be translated to the proper position. If the cone is positioned on the other end point, it may be obscured and thus invisible. So it should be positioned before the intersection of the line and surface of the Shape3D. To find the intersection, the binary search method is employed.

Arrow Behavior

After the scene is drawn, a user can navigate and interact with the scene. These functions are implemented by Java3D Behavior object. PickTranslateBehavior, PickRotateBehavior, and PickZoomBehavior facilitate the interaction with the scene graphical entities while KeyNavigatorBehavior carries out the navigation within the scene. In order to make the arrows follow the movements of the entities, which they are pointing from and to, when a user moves, zooms, or rotates the entities, ArrowBehavior was implemented. Each time a graphical artifact is moved, the arrows that are attached to it will first be detached, and then new arrows according to the change will be attached to replace the old ones. WakeupOnTransformChange class specifies a wakeup whenever the transform within a specified TransformGroup changes. PickDoubleClickBehavior is designed for the class diagram to detect the operation of double-click. A user can get the detailed information of a class such as its attributes, methods, coupled classes, etc. by double-clicking on the selected class. PickDoubleSysBehavior is implemented for the system diagram to provide the user with the ability to view only the sub class diagram that only contains the classes within one package in the system diagram.

- *Graph drawing layout algorithms* [11]

  There exist many constraints for layout algorithms, such as the nodes and edges must be evenly distributed, edge-crossing should be kept to a minimum, nodes with equal depth should be placed on the same horizontal line, the distance between sibling nodes is usually fixed, etc. Aesthetic rules are also imposed on the graph drawing layout. Usability studies conducted in order to evaluate the relevance of these aesthetics for the end-user show that reducing the crossings is by far the most important aesthetic than others such as maximizing symmetry, etc [11].

There are two other important issues related to layout algorithms. One is predictability, the other one is time complexity. Predictability is also referred to as preserving the mental map of the user. That is to say two different runs of the algorithm should lead to the same visual representations. Time complexity of a layout algorithm indicates the time required to compute the optimized layout. For the layout algorithm to be applicable, the time required to compute a layout should be bound, preferable it should be as close as possible to real-time. However, depending on the number of entities, the given screen space and the number of edges, that have to be optimized, the computation time and therefore the time complexity might be unbound. [11].

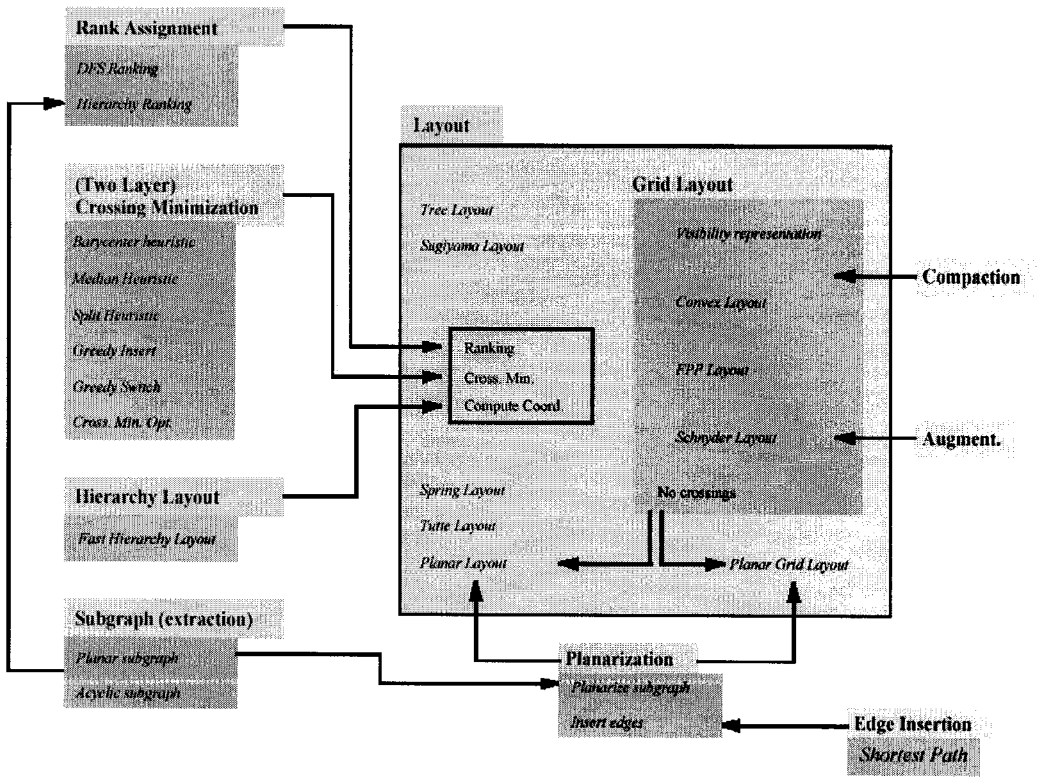


Figure 31 Overview of graph layout algorithms [11]

In Figure 31, an overview of the graph layout algorithm is presented. The tree layout contains a variety of layouts. Classical tree layout algorithms produce top-down or left-to-right tree layouts. H-tree layout, as shown in Figure 32A, is adapted to balanced binary trees. The radial view layout (see Figure 32B) positions nodes on concentric circles according to their depth in the tree. The balloon view (see Figure 32C) uses a cone tree algorithm and projects it onto the plane [11].

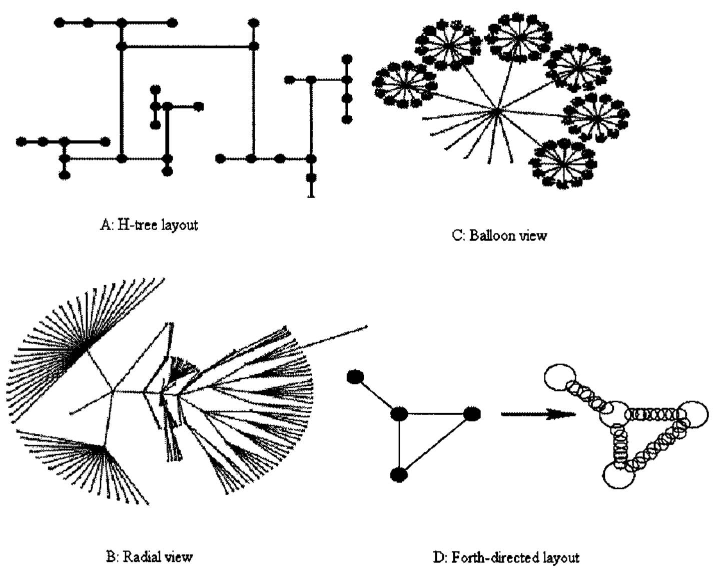


Figure 32 Graph Layout Examples [11]

Force-directed algorithms, which are also termed as spring layout (see Figure 32D), use a physical analogy to draw graphs which make it easier to understand and relatively simple to code. Edges are modeled as springs, and vertices are equally charged particles which repel each other.

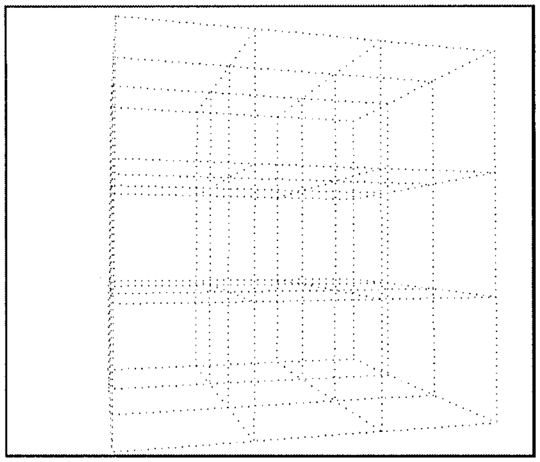


Figure 33 Grid Layout [40]

Within the *CONCEPT* project, a grid layout was implemented and further improved using hill-climbing algorithm to minimize the edge-crossings. The grid layout algorithm positions nodes of a graph at points with integer coordinates (see Figure 33). State-space searching begins from the initial-states until it reaches a predefined goal-state [37]. The goal here is edge-crossing minimization. There are two basic searching methods: blind search and heuristic search. A heuristic search is used when the problem has ambiguities or no exact solution. In these cases, the heuristic search can provide an acceptable solution. Among the heuristic searches, the hill-climbing algorithm was implemented within the *CONCEPT* project. The hill-climbing algorithm is very good in exploiting their neighborhood. Relationships between entities are most important concerns for hierarchical data structure. The improved grid layout employed here aims at generate the graphical representations of minimum edge crossings [40].

- *Program slicing* [54]

Large systems provide major challenges to the visualization techniques and their scalability as well as their usability [5]. Slicing is one method to reduce the visual overhead by allowing for a reduction in the number of entities to be displayed. The combination of visualization and program slicing can help the maintainers focus on the interested items [48]. The information reduction reduces a programmer's cognitive load during the program comprehension process. Program slicing is a valuable method to restrict the focus of a task to specific sub-components of a program.

The notion of static program slicing originated in the seminal paper by Weiser [54], who defined a slice S as a reduced, executable program obtained from a program P by removing statements such that S replicates parts of the behavior of P. In Weiser's approach, which is based on program dependencies, slices are consecutive sets of indirectly relevant statements. A static program slice consists of these parts of a program P that potentially could affect the value of a variable v at a point of interest. The static algorithm uses only statically available information for the slice computation; hence this type of program slicing is referred to as a static slice. Another major extension of program slicing was introduced by Korel and Laski [54] called dynamic slicing. For the dynamic slicing approach, not only static information is available, but also dynamic information regarding the program execution on some program input. The dynamic slicing approach emphasizes its dependence on program execution. 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 the program terminates. The reason for this diversity of slicing types and methods is the fact that different applications require different properties of slices. Slicing has been shown to be useful in program debugging, testing, program comprehension, and software maintenance [54].

There are three phases in *CONCEPT* architecture as discussed above. Program slicing is part of the second, the analysis phase. The slicing result can be used to reduce the number of entities to be displayed at a time. For example, the Swingset library used for the *UML3D* tools contains 204 classes in 10 packages and 597 relations between them presented in the class diagram in *UML3D* for Swingset system (see Figure 34). Even for such a relative small software system, some of the entities shrink to pixel size and some of the important information might be obscured by the unconcerned entities and cannot be easily grasped from the presentation at once. In these situations, program slicing might be applied to reduce the complexity of the graph by reducing the number of entities to be displayed (depending on the slicing criterion). Typical reduction achieved by program slicing is somewhere between 50-80%. The advantage of the slicing based filtering of the information is that the slice contains only information relevant with respect to the particular slicing criterion. Program slicing can also be applied for the other diagrams supported by *UML3D*, e.g. system diagrams, collaboration diagrams, etc.

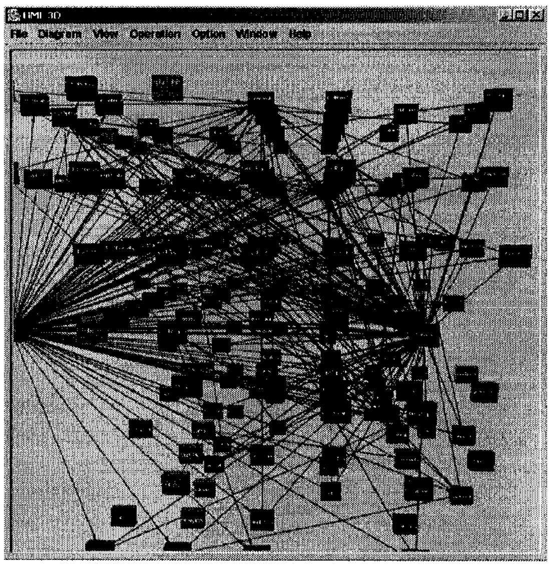


Figure 34 Class diagram for Swingset before slicing

- *UML3D GUI*

The Abstract Window Toolkit (AWT) provides the user interface for Java program. AWT relies on "peer-based" rendering to achieve platform independence. But subtle difference in platforms resulted in inconsistent look-and-feel, and platform-dependent bugs. Swing avoids the problems by using a non-peer-based approach; with the result it may be slower than AWT. "Swing" refers to the new library of GUI controls (buttons, sliders, checkboxes, etc.) that replaces the somewhat weak and inflexible AWT controls.

Lightweight popup windows are more efficient than heavyweight (native peer) windows, but lightweight and heavyweight components do not mix well in a GUI. If an application mixes lightweight and heavyweight components, lightweight popup should be disabled by adding code in the program like: JPopupMenu.setDefaultLightWeightPopupEnabled(false). Some look and feels might always use heavyweight popup, no matter what the value of this property.

4.2.5 *UML3D* – visualization environment

The *UML3D* framework implements both 2D and 3D metaphors for software visualization. Texture, abstraction mechanism, and user interfaces are employed in the representation. The form of representation needs to be best conveying the target information to the user which depending on the goals and target of the software visualization systems, the type of the users, and available medium [36].

- *Abstraction at different levels and context switch*

UML3D supports program comprehension at different levels of abstraction and allows for context switching among these different views. Figure 35 shows the menu associated with the different diagram types.

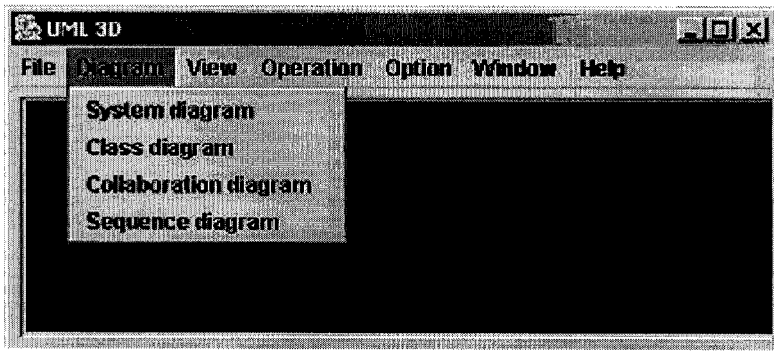


Figure 35 Menu selections for abstractions at different level

The system diagrams show the higher level abstraction – in the form of packages. The class diagrams show complete system on the class level. Classes in the same package are kept in the same color. The different relationship types among classes can be identified by the different arrow styles. In each diagram, the names of the package, the class, and the object are mapped as text attached on the surface of the graphical entities.

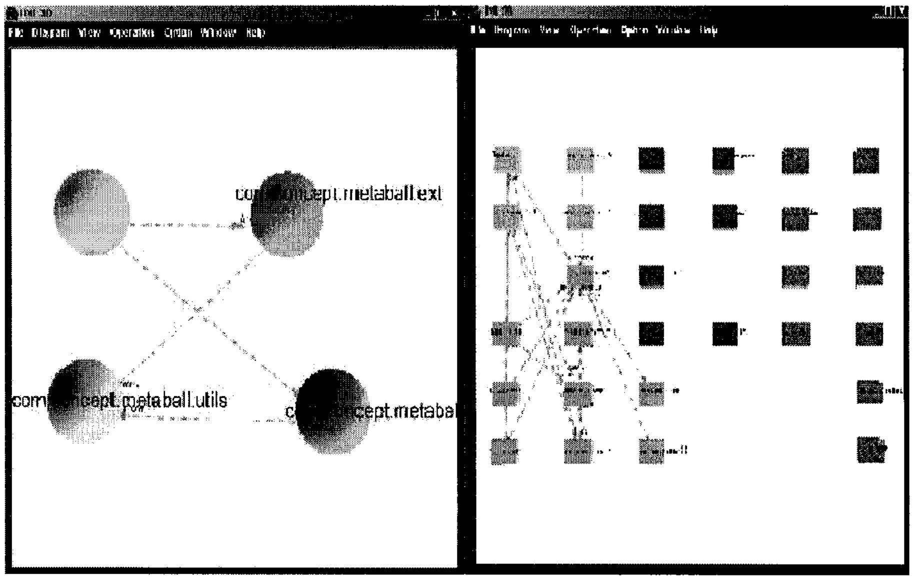


Figure 36 2D System and Class Diagrams in *UML3D*

Through selecting and double clicking a graphical entity one can access the specific information of it (details-on-demand). When double-click on a graphical entity in the class diagrams, the detailed information of that class such as package name, attributes and methods will be shown in another popup window.

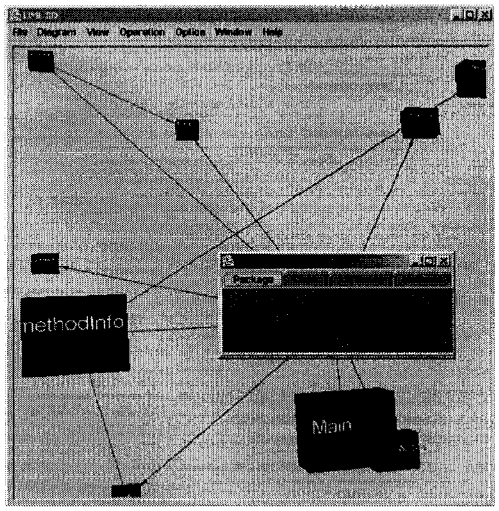


Figure 37 Details of a Class

- *Static and dynamic information*

UML3D provides both static and dynamic information of a software system. Within the *CONCEPT* project, dynamic program execution is recorded in the form of execution trace. The execution trace is generated by *INSTR* and stored in *PostgreSQL* database. *INSTR* is a Java package used for instrumenting Java source code. For example, it can be used for tracing of individual source lines. *INSTR* accomplishes the task by means of six steps: 1) parse the source code into a tree, 2) annotate the parse tree with instrumentation, 3) write the annotated tree to a file, which will contain Java source code with some additions, 4) compile the annotated source, 5) execute the program and collect instrumentation data, and 6) remove the annotations from the source code [55]. This recorded information can be used to visualize the run-time behavior of the system. It has been shown that during the coding phase a programmer is actually thinking not of the code itself, but of its execution behavior [49]. When a programmer is debugging, the programmer is first trying to get the reverse mapping:

from undesired behavior to the portion of code that causes it and what the code must look like to evoke the desired behavior. Sometimes coders will have to make small changes to the behavior without having any knowledge of which part of the code does what. Note that the programmer only needs to know which part of the code implements the portion of behavior that needs to be changed. For the visualization of dynamic program behavior, *UML3D* introduces the notion of dynamic sequence and collaboration diagrams. Both of these diagrams are extension of the traditional counterparts in UML. Rather than conveying static (structural) information in the UML versions, these dynamic versions utilize additionally the recorded execution information to show the program behavior. In *UML3D*, the sequence diagram shows the runtime behavior, by not only visualizing the timely sequence of the executions, but also the interaction among the different entities.

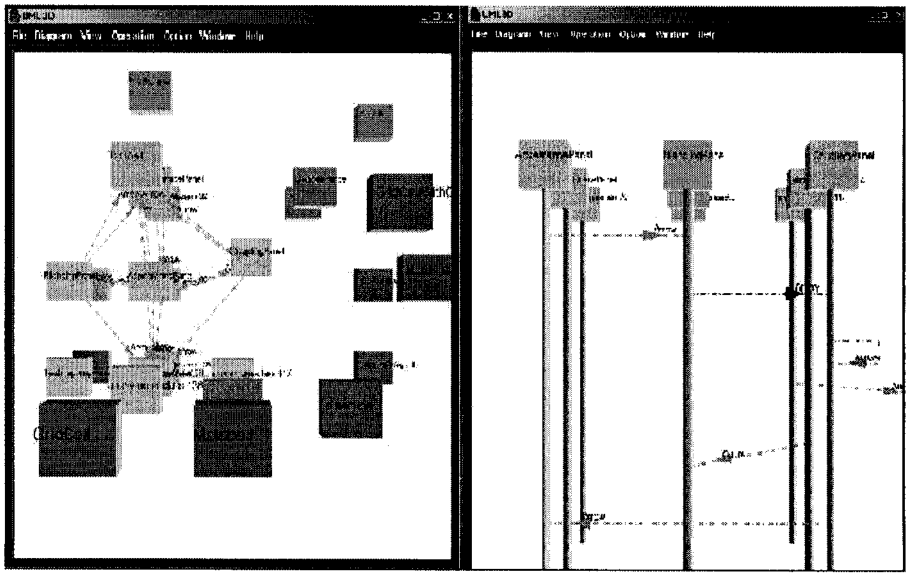


Figure 38 3D Static and Dynamic Diagrams in *UML3D*

- *Navigation and interactions*

In *UML3D*, intuitive navigation and interactions are added. One of the key requirements for a successful adoption of a tool is not only its ability to provide an intuitive navigation trough the diagrams and their information, but also to provide an intuitive way of interaction with the views. In *UML3D* the user can use either the keyboard to navigate or move within/among the views. Additionally to the keyboard, the users also can use mouse to move, rotate, and zoom in/out either the whole view or selected graphical entity.

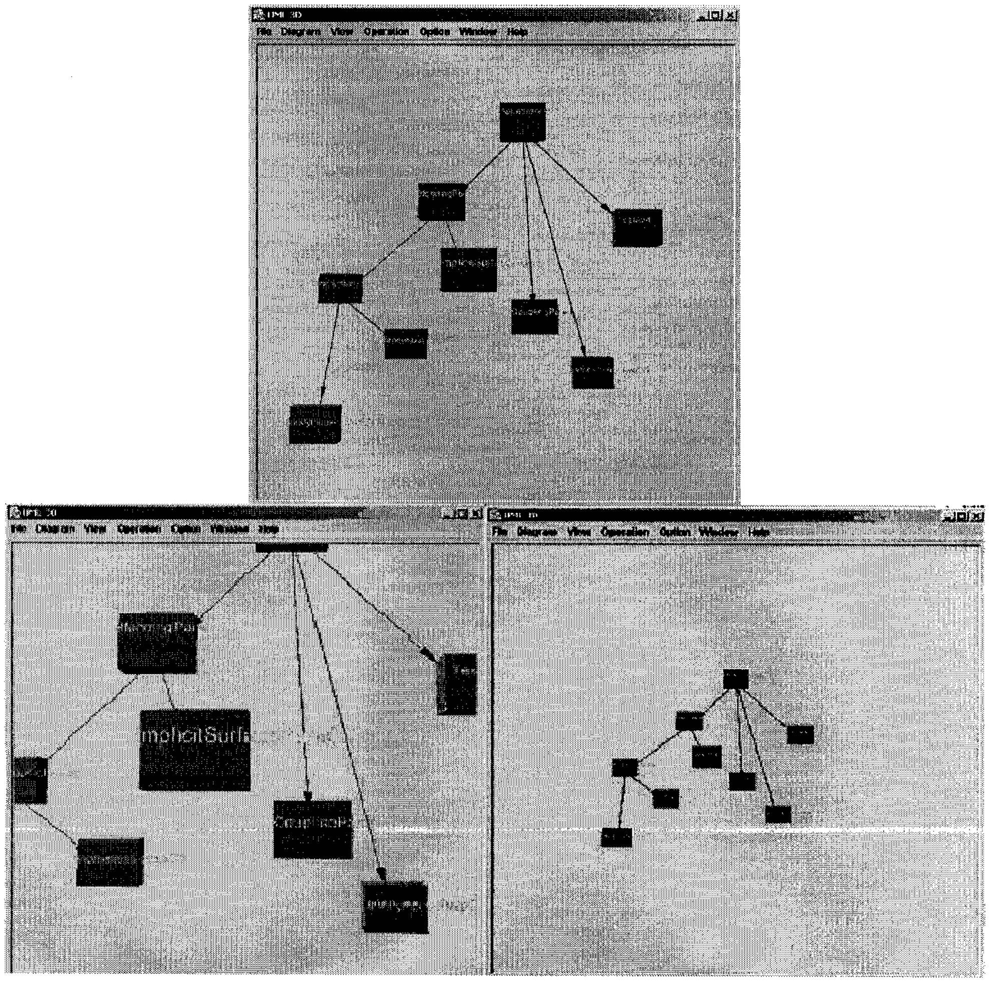


Figure 39 Zoom in/out of a diagram

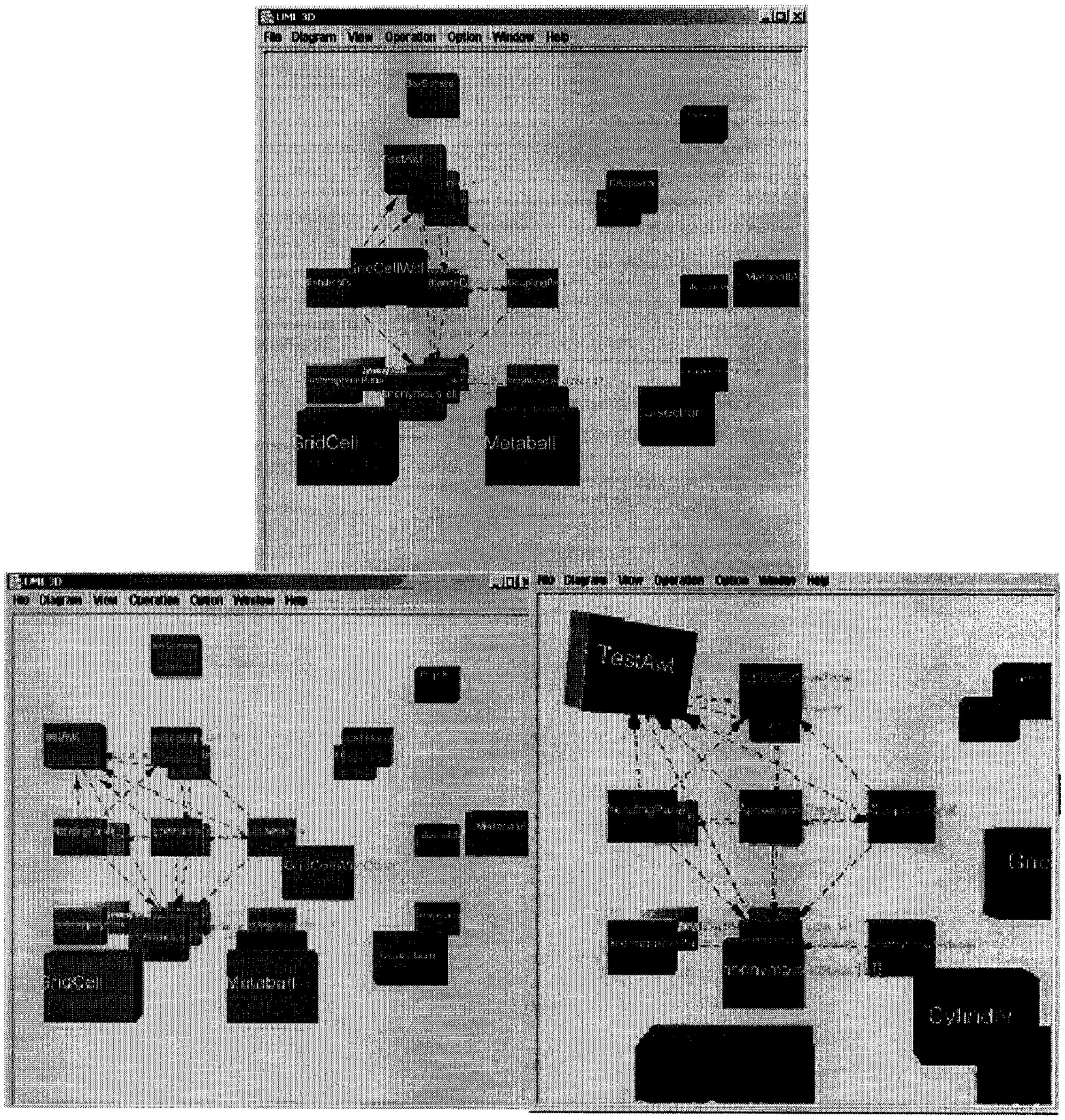


Figure 40 Move and rotation of a diagram

- *Filtering*

When double-click on a graphical entity in the system diagrams, which represents a package, the classes only in that package will be represented (See Figure34).

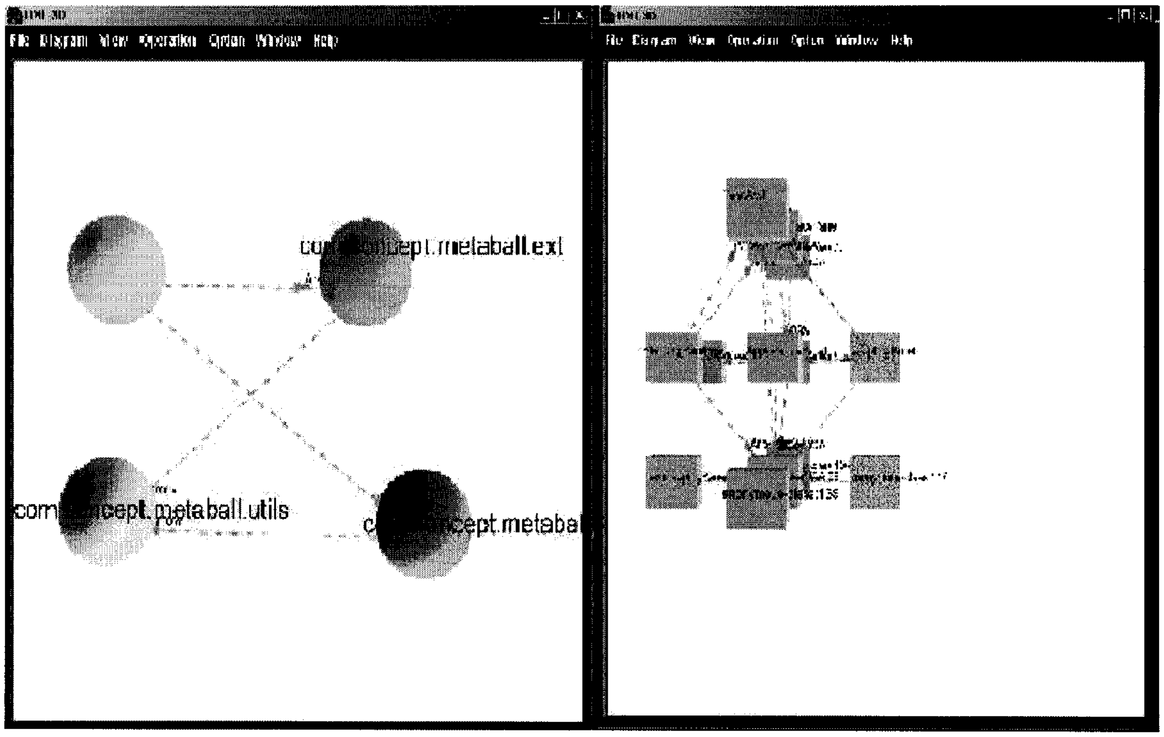

UML 3D
File Diagram View Operation Option Window Help
com.concept.metaball.ext
com.concept.metaball.utils
com.concept.metaba

Figure 41 3D System and Specific Class Diagrams in *UML3D*

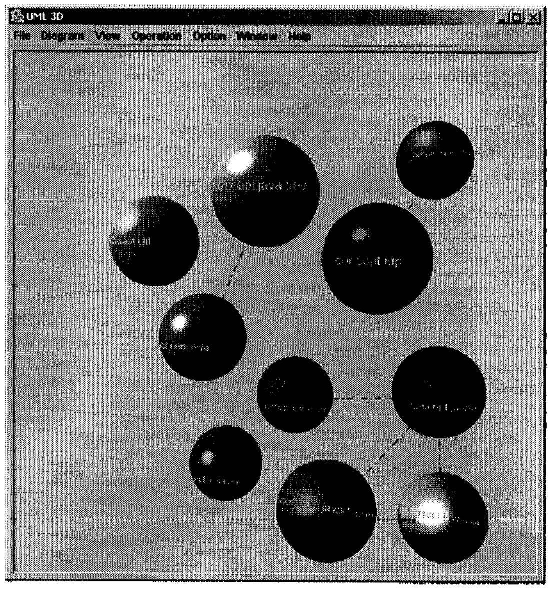


Figure 42 System diagram for Swingset

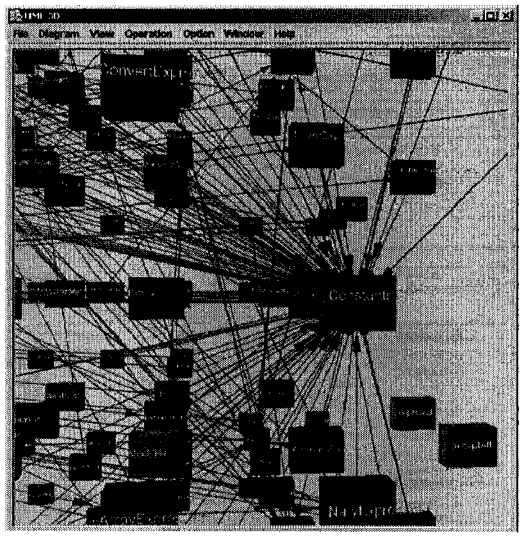


Figure 43 Class diagram before filtering

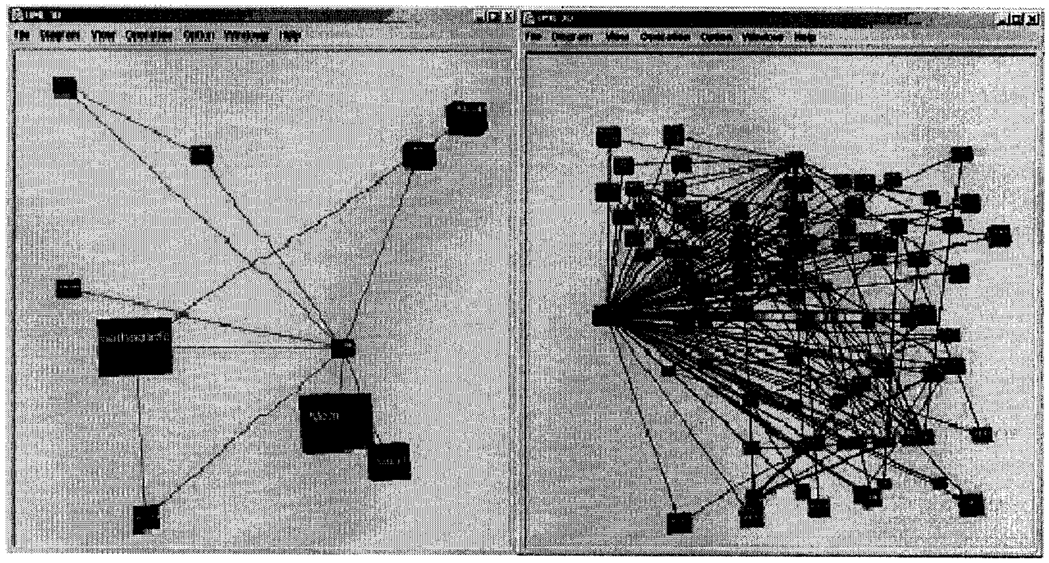


Figure 44 Class diagram after filtering

- *History record*

The provision of a history options, provides support for undo, replay, saving and review of previously performed action, providing the user with additional options of maintaining a consistent and repeatable view of the system.

Chapter 5 Initial Usability Study

A large number of software visualization tools have been proposed to address different problems in comprehension and visualization of software systems [22]. However, far less effort has been devoted to evaluating existing software visualization tools. As a result, it is unclear how effective these tools are and how they benefit the user of these tools during a particular comprehension task. Lessons learnt in successive designs of software visualization tools can be used in motivating future designs which in turn could be compared and improved. Empirical evaluations are needed to improve the design and implementation for existing and new software visualization tools development [18].

- *Evaluation criteria*

  The evaluation can be done according to the following issues [18]:

  Usefulness: The goal of software visualization is to make it easier for the users to understand the programs under study using graphical representations. So whether the visualization really facilitates maintenance and aids the maintainers form mental models about the systems must be taken into consideration.

  Intuitiveness: Is the visualization easy to understand? Cognitive issues have been receiving increased attention by people who study the program comprehension process. The software visualization tools are used to reduce the cognitive load for the users. Thus, the intuitiveness of the visualization tools should be an important issue when evaluating the software visualization tools. Both the mapping from the software artifacts to graphical entities and the manipulation of the representation should be intuitive.

Scalability: As the scale of software systems increases, the handling of large software systems is important to the software visualization tools. Currently, few software visualization tools, both academic and enterprise, can handle large, real-world systems. Does the visualization work well for these systems? To make the tools more useful and widely accepted, the scalability of the tools cannot be ignored. E.g. the complexity of the improved grid layout algorithm used in this project is O (n^3) [40]. The layout algorithm works well for small systems like elevator program, etc. However, when it deals with very large systems like Swingset with hundreds of classes, its performance is reduced greatly.

Except for the criteria mentioned above, there are many other issues that cannot be ignored, for example, if the speed of the software is fast enough, etc.

- *Questionnaire design*

SUMI (Software Usability Measurement Inventory), which has a list of 50 questions, is used for general software usability evaluation [51]. Based on the evaluation criteria for software visualization tools discussed above such as usefulness, intuitiveness, and scalability, ten questions are selected from SUMI (see Appendix B) and the preliminary test of the system is performed by seven people in *CONCEPT* Research

Group.

- *Data collection*

Data collection can be done through electronic mailing lists, newsgroups, or known users of specific software visualization tools. As this project is in the process of developing and fulfillment, the preliminary evaluation task is done in the research group of *CONCEPT.* Seven people tested the visualization tool and answered the selected questions referenced from *SUMI* [22].

- *Data analysis*

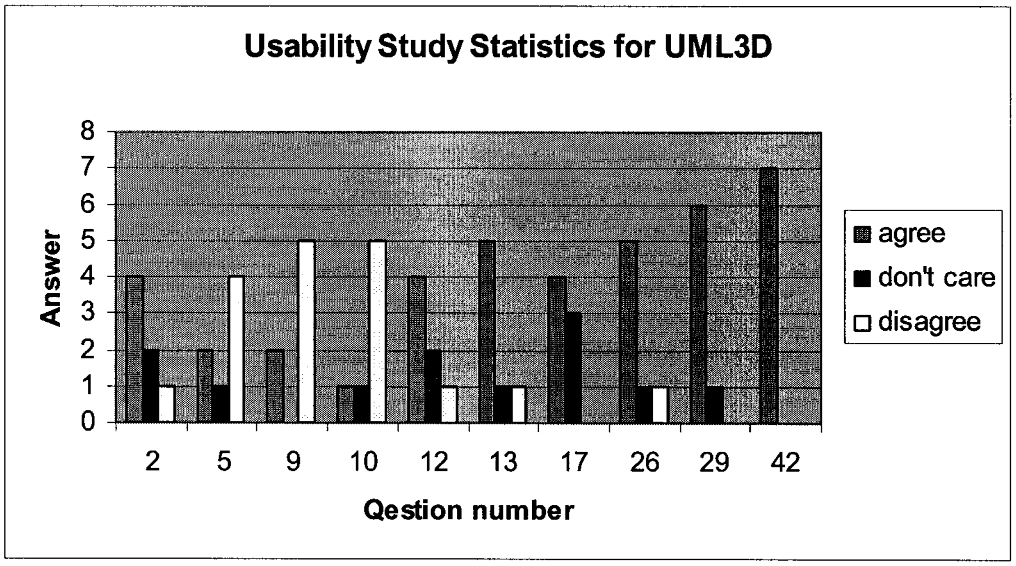


Figure 45 Initial Usability Study Results

Data analysis is done based on the collected data. The results (see Figure 45) of our initial usability study show that all the seven people who tested the software agree that the system has a very attractive presentation (question 42). About 60% of them feel that working with this software is mentally stimulating and satisfactory that they would recommend this software to their colleagues (question 2, 12, and 17). More than 70% people think that this software is easy to use and the information is presented in a clear and understandable way (question 5, 9, 10, 13, 26, and 29).

Besides the positive feedback of UML3D, there are some other criticisms for the system. Some of the negative feedbacks are about the navigation disorientation, which are mentioned in the challenges of 3D. Some other criticisms are related to the interface of the framework, such as the lack of progress bar, wizard window, shortcut, etc. The refinement of the interface is not much considered in current stage because the project is in the procedure of functionality fulfillment and there will be identical interface for all software visualization tools in *CONCEPT*. The speed of the visualization system for larger software is also complained by some of the users. Layout is a bottleneck for the visualization pipeline. This issue should be taken into consideration for future improvement of the system to enhance the system's performance and make it more acceptable. Last, the 3D UML' visibility is challenged by the overwhelming amount data to be presented at a time. Some of the users suggest that there should be a strategy for information query, which will simplify the graph by only showing the software artifacts that have connection with the interested entity. This issue has already been pointed out in the future research like customized query.

The results of our initial usability study show that *UML3D* can improve on UML and provide software engineers with additional guidance and intuitive visualization approaches that support the software comprehension process. The results of the study will be an important factor for the future evolution of *UML3D* and its integration with other software visualization tools, such as *MetaVis* and *JVC*, in *CONCEPT*.

Chapter 6 Conclusions and Future Research

Following the conclusions, this chapter will include the discussion on the future research.

6.1 Conclusions

One of the major motivations of this research was to improve on traditional 2D UML techniques to improve the scalability, usability and navigability. The presented work (*UML3D*) applied 3D techniques to UML to take advantages of 3D space and the additional features that can be applied in the 3D space. *UML3D* addressed some other shortcomings of UML by providing intuitive navigation and interactions with the diagrams, combining source code analysis like program slicing, etc. The main contribution of the presented research can be found in the following areas:

- Applying and mapping traditional 2D UML notation to the 3D space.
- Investigation and implementation of different mapping approaches between software artifacts and their possible representations in the 3D space.
- Combining structural information of the traditional UML diagrams with filtering techniques (e.g. program slicing, selective zoom, etc) to reduce the complexity of the diagrams and to improve the scalability of the visualization techniques.

- The diagrams not only provide the static structural visualization (based on the source code), but also include dynamic/behavioral information (based on the execution trace), provided by the *CONCEPT* database.

- Integration of a self-organizing layout algorithm that can be applied to both traditional 2D and the presented 3D representation.

- Extension to the Java 3D library, supporting additional drawing objects, like arrows.

- Allowing for easy context switching among different visualization techniques and views.

- Provided a complete implementation of *UML3D* and its integration to the *CONCEPT* framework in Java 3D.

- Performed an initial usability study to gain some feedback with respect to the applicability.

However, it should also be noted, that the current implementation of the *UML3D* tool has still several limitations. One drawback in *UML3D* is that the layout algorithm, which works well for small systems, cannot handle large systems efficiently. For larger number of entities the layout optimization might cause a significant performance bottleneck. For larger systems, we have currently imposed maximum time limit of 120 minutes before the layout optimization will be terminated. This significant computation overhead also makes a real-time layout optimization impossible. Furthermore, scalability issues, in particular for the behavior (sequence and collaboration) diagrams remain unsolved and these challenges will have to be addressed in future work.

6.2 Future Research

According to the evaluation results of the existing software visualization tools and lessons learned from it, further research and improvements in the domain can be done in the following aspects:

- *Customized query*

  Few of the efforts to enhance and facilitate software understanding have been really successful and helped develop tools in active use today. The software visualization systems, which intuitively seemed so obviously useful, were not being used [25]. Why are experts often resistant to other people's visualization? The reasons are: 1) the failure to address the actual issues that arise in software understanding, 2) the difficulty in such systems when setting up the data for them, and 3) when understanding how to get the desired information out of the overwhelming amount of information [6]. The problem can be solved by providing visual query language for specifying what information should be visualized [25]. User-centered approaches should be employed. As an example, BLOOM is a system providing static/ dynamic and 2D/3D software visualization with a visual query language. It is based on an object-oriented view with an entity-relationship based query language. The system provides facilities for saving useful queries so they can be reused.

- *Searching*

Searching is one of the two navigation styles that has long been available on program code, but is largely absent in architecture visualization tools. Searching employs a bottom-up comprehension strategy and is needed to reveal the facts that were used to build the concepts [19]. Moreover, semantic searches, which are more useful, are seldom supported by the software visualization tools [19].

- *Automation*

The degree of automation is also very important for a software visualization tool. An effort should be made to increase degree of automation and decrease as much as possible user involvement in comprehending, maintaining and dealing with the source code itself [39].

- *Language independent*

The software visualization tools available always have language constraints. The ideal software visualization tools should be language independent and flexible at the data format.

- *Integration*

All visualization tools must have separate key parts such as parser, layout engine and interface. The integration with other tools can make the software visualization tools more acceptable and gain more feedback.

- *Evaluation*

Empirical evaluation can be gained through users' feedback and experts' study. Many software visualization tools are only developed and used in academia [18]; however, few of these are evaluated and little is learned about their worth.

Software visualization is a rapidly developing field. Efforts made in this field have shown promising results. The application of 3D and virtual reality techniques gives a new life to software visualization. However, 3D is not a cure-all and encounters many challenges as well. There is still a long way for future research in this field.

Bibliography

[1] Mary Ellen Foster, "Automatically generating text to accompany information graphics", Master Thesis, Department of Computer Science, University of Toronto, July 1999.

[2] C. Knight, "Virtual Software in Reality", PhD Thesis, Department of Computer Science, University of Durham, June 2000.

[3] Juergen Rilling, "Lecture Notes: Reverse Engineering", Internet Document, http://www.cs.concordia.ca/~teaching/comp691l/html/lecture_notes.html, August 2003.

[4] Ramage, M., and Bennett, K. H. "Maintaining Maintainability," Proceedings of the

International Conference on Software Maintenance, Washington, 16-20 November

1998.

[5] K. B. Gallagher and L. O'Brie, "Reducing visualization complexity using decomposition slices", in SoftVis'97, The 1997 Software Visualization Workshop, Adelaide, Australia, Dec 1997.

[6] Margaret-Anne Storey, "Software Visualization", http://www.cs.uvic.ca/~mstorey/ teaching/infovis/course_notes/softviz.pdf, August 2003.

[7] S. Robitaille, R. Schauer, and R.K. Keller, "Bridging Program Comprehension Tools by Design Navigation", Proc. of the Intnl. Conf. on Software Maintenance: 22-32, October 2000.

[8] M. -A.D. Storey, F.D. Fracchia, and H.A. Muller, "Cognitive Design Elements to Support the Construction of a Mental Model during Software Exploration", Journal of Software Systems, 44: 171-185, 1999.

[9] Anuar Musa, "UML By Examples", http://www.geocities.com/SiliconValley/ Network/1582/uml-example.htm, August 2003.

[10] C. Knight, "Visualisation for program comprehension: information and issues", Technical Report, Department of Computer Science, University of Durham, 1998.

[11] I. Herman, G. Melancon, and M. Marshall, "Graph visualization and navigation in information visualisation: a survey", IEEE Transactions on Visualization and Computer

Graphics, 6(1):24-43, 2000.

[12] Guardian Education Interactive Ltd., "Pyramids of numbers", http://www.learn.co.uk/default.asp?WCI=Unit&WCU=2400, August 2003.

[13] Nurse Bob, "Cardiology In Critical Care", http://rnbob.tripod.com/electroc.htm, August 2003.

[14] S. Decker, J. Jannick, S. Melnik, P. Mitra, S. Staab, R. Studer, and G. Wiederhold, "An Information Food Chain for Advanced Application on the WWW", Proc. 4th European Conf. on Research and Advanced Technology for Digital Libraries (ECDL'00), LNCS 1923, p. 490 ff., 2000.

[15] Oren Etzioni, "Moving Up the Information Food Chain: Deploying Softbots on the World Wide Web", Proceedings of the 13th National Conference on Artificial Intelligence and the 8th Innovative Applications of Artificial Intelligence Conference , AI Magazine, 18(2): pp.11-18, Intelligent Systems on the Internet, 1997.

[16] Juergen Rilling and Ahmed Seffah, "A Task-Driven Program Comprehension Environment", COMPSAC 2001: 77, 2001.

[17] Jonathan I. Maletic, Andrian Marcus, and Michael L. Collard, "A Task Oriented View of Software Visualization", Department of Computer Science, Kent State University, http://citeseer.nj.nec.com/correct/547223, August 2003.

[18] Sarita Bassil and Rudolf K. Keller, "A Qualitative and Quantitative Evaluation of Software Visualization Tools", In Proceedings of the Workshop on Software Visualization, pp. 33-37, Toronto, ON, May 2001.

[19] Susan Elliott Sim, Charles L.A. Clarke, Richard C. Holt, Anthony M. Cox, "Browsing and Searching Software Architectures", Proceedings of the International Conference on Software Maintenance, pp. 381-390, Oxford, England, September 1999.

[20] Erez Hartuv, Ron Shamir, "A Clustering Algorithm based on Graph Connectivity", Technical report, Department of Computer Science, Tel Aviv University, 1999.

[21] S.S. Alhir, "Extending the Unified Modeling Language (UML)", Internet Document,

http://home.earthlink.net/~salhir/ExtendingTheUML.PDF, 1999.

[22] Margaret-Anne D. Storey, "A Cognitive Framework for Describing and Evaluating Software Exploration Tools", PH. D. Thesis, School of Computing Science, University of Victoria, 1998.

[23] Software Engineering Group, "The FUJABA Environment", Department of Mathematics and Computer Science, University of Paderborn, http://www.uni-paderborn.de/cs/fujaba/main.html , August 2003.

[24] Hausi A. Müller, Jens H. Jahnke, Dennis B. Smith, et al., "Reverse Engineering: A Roadmap", In A. Finkelstein, editor, The Future of Software Engineering, ACM Press, 2000.

[25] Steven P. Reiss, "An Overview of BLOOM", ACM SIGPLAN-SIGSOFT, pp. 2-5, ACM Press, New York, USA, 2001.

[26] Brian Johnson and Ben Shneiderman, "Treemaps: a space-filling approach to the visualization of hierarchical information structures", IEEE Visualization '91, pp. 284-291, IEEE CS, 1991.

[27] Mark Bruls, Kees Huizing and Jarke J. van Wijk, "Squarified Treemaps", Joint Eurographics and IEEE TCVG Symp. on Visualization (TCVG 2000), pp. 33-42. IEEE Press, 2000.

[28] Sarita Bassil and Rudolf K. Keller, "Software Visualization Tools: Survey and Analysis", 9th International Workshop on Program Comprehension, IEEE, Toronto, Ontario, Canada, May 2001.

[29] Shaffer, C.A., Heath, L.S., and Yang, J., "Using the Swan Data Structure Visualization System for Computer Science Education", Proc. SIGSCE '96, Philadelphia PA, February 1996.

[30] Terry Quatrani and Grady Booch, "The Modeling with Rational Rose 2000 and UML", Addison Wesley Professional, 1998.

[31] Maletic, J.I., Leigh, J., and Marcus, A., "Visualizing Software in an Immersive Virtual Reality Environment", ICSE'01 Workshop on Software Visualization, pp.49-54, Toronto, CA, May 13-14, 2001.

[32] Klaus Alfert and Alexander Fronk, "Manipulation of 3-dimensional Visualizations of Java Class Relations", Proceedings of the 2002 IDPT Conference - The Sixth World Conference on Integrated Design & Process Technology, Pasadena, CA, June 2002.

[33] Peter Young, "Software Visualization", http://vrg.dur.ac.uk/misc/PeterYoung/pages/work/documents/index.html, August 2003.

[34] Yamin Wang, Wei-Tek Tsai, Xiaoping Chen, and Sanjai Rayadurgam, "The Role of Program Slicing in Ripple Effect Analysis", www-users.cs.umn.edu/~wangy/./paper/psinrea.ps, August 2003.

[35] John Lamping, Ramana Rao, and Peter Pirolli, "A Focus+Context Technique Based on Hyperbolic Geometry for Visualization Large Hierarchies", Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems, pp. 401--408, ACM Press, 1995.

[36] Andrian Marcus, Louis Feng, and Jonathan I. Maletic, "3D Representations for Software Visualization", Proceedings ACM 2003 Symposium on Software Visualization,

pp. 27-36, 207-208, San Diego, California, USA, June 11-13, 2003.

[37] Juergen Rilling and Sudhir P. Mudu, "On the Use of Metaballs to Visually Map Source Code Structures and Analysis Results onto 3D Space", 9th Working Conference on Reverse Engineering (WCRE'02), pp.209-308, Richmond, Virginia, USA, October 29 - November 01, 2002.

[38] Neil Trevett ,"Challenges and Opportunities for 3D Graphics on the PC", Presentation for SIGGRAPH Eurographics Hardware Workshop in Computer Graphics, http://www.ibiblio.org/hwws/previous/www_1999/presentations/keynote/sld001.htm, August 2003.

[39] P. Mulholland, "Using a Fine-grained Comparative Evaluation Technique to Understand and Design Software Visualization Tools", 7th Workshop on Empirical Studies of Programmers, Alexandria, VA, USA, October 24- 26, 1997.

[40] J. Wang, "MetaViz – Issues in Software Visualizing Beyond 3D", Master Thesis, Department of Computer Science, Concordia University, August 2003.

[41] Juergen Rilling, "CONCEPT", http://www.cs.concordia.ca/CONCEPT/index.shtml, August 2003.

[42] Peter Grogono, "Concordia University Graphics Library", http://www.cs.concordia.ca/~faculty/grogono/CUGL/cugl.html, 2002.

[43] J. Conklin, "Hypertext: An Introduction And Survey", IEEE Computer, 20(9): 17-41, 1987.

[44] Warwick Irwin and Neville Churcher, "XML in the Visualization Pipeline", Research and Practice in Information Technology, Sydney, Australia, 2001.

[45] Mao Lin Huang, Peter Eades, and Junhu Wang, "Online Animated Graph Drawing Using a Modified Spring Algorithm", Australian Computer Science Comm.: Proc. 21st Australasian Computer Science Conf., 1998.

[46] Sun Microsystems Java 3D Engineering Team, "The Java 3D™ API Specification", http://java.sun.com/products/java-media/3D/forDevelopers/J3D_1_2_API/j3dguide/, August 2003.

[47] Sun Microsystems Java 3D Engineering Team, "Getting Started with the Java 3D™ API", http://developer.java.sun.com/developer/onlineTraining/java3d/, August 2003.

[48] Andrea de Lucia, "Program Slicing: Methods and Applications", 1st IEEE International Workshop on Source Code Analysis and Manipulation (Florence, Italy, 2001), pp. 142-149, IEEE Computer Society Press, Los Alamitos, California, USA, 2001.

[49] Manos Renieris and Steven P. Reiss, "Almost: Exploring Program Traces", NPIVM'

99 Workshop, pp. 70—77, Kansas City, MO, November 6, 1999.

[50] J. Stasko et al, "Software Visualization: Program as a Multi-Media Experience", MIT Press, 1998.

[51] Human Factors Research Group, "SUMI", http://www.ucc.ie/hfrg/questionnaires /sumi/index.html, August 2003.

[52] SGI, "3D File System Navigator for IRIX 4.0.1+", http://www.sgi.com/fun/ freeware/3d_navigator.html, August 2003.

[53] W3C, "XSL Transformations (XSLT)", Internet Document, http://www.w3.org/TR/ xslt, August 2003.

[54] Juergen Rilling, "Investigation of Program Slicing and its Applications in Program Comprehension of Large Software Systems", PhD Thesis, Department of Computer Science, Illinois Institute of Technology, July 1998.

[55] Glen McCluskey & Associates LLC, " Java(tm) Source Code Instrumentation ", Internet Document, http://www.glenmccl.com/instr/instr.htm, August 2003.

[56] Arthur Tateishi, "Filtering Run-Time Artifacts Using Software Landscapes", Master Thesis, Department of Computer Science, University of Toronto, 1994.

[57] Michael Fotta, Ben DeLong, Matt McMahon, and Phillip Merritt, "Software Visualization for IV&V", Software Assurance Symposium (SAS) 2002, September 6, 2002.

Appendix A: Example of Data in XML Format for Static Diagrams of *UML3D*

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Created by gary on April 26, 2003, 1:10 PM --><entity_relation>
<entities>
    <entity id="0" name="GridCell" color="METAL" package="concept.javad">
        <size factor="1" controlRadius="1"/>
        <coordinator x="1" y="3" z="2"/>
    </entity>
    <entity id="1" name="LabledMetaball" color="PEWTER" package="concept.java">
        <size factor="1" controlRadius="1"/>
        <coordinator x="2" y="3" z="1"/>
    </entity>
    <entity id="2" name="Cylinder" color="PEWTER" package="concept.java">
        <size factor="1" controlRadius="1"/>
        <coordinator x="1" y="2" z="1"/>
    </entity>
    <entity id="3" name="StripCylinder" color="PEWTER" package="concept.java">
        <size factor="1" controlRadius="1"/>
        <coordinator x="1" y="1" z="1"/>
    </entity>
    <entity id="4" name="Propeller" color="PEWTER" package="concept.java">
        <size factor="1" controlRadius="1"/>
        <coordinator x="1" y="3" z="1"/>
    </entity>
    <entity id="5" name="CText2D" color="GOLD" package="concept.java.tree">
        <size factor="1" controlRadius="1"/>
        <coordinator x="0" y="1" z="1"/>
    </entity>
    <entity id="6" name="BoxSphere" color="GOLD" package=" concept.java.tree">
        <size factor="1" controlRadius="1"/>
        <coordinator x="1" y="1" z="0"/>
    </entity>
    <entity id="7" name="Metaball" color="METAL" package="concept.idp">
        <size factor="1" controlRadius="1"/>
        <coordinator x="0" y="1" z="0"/>
    </entity>
    <entity id="8" name="GridCellWithColor" color="METAL" package="sun.asm">
        <size factor="1" controlRadius="1"/>
        <coordinator x="1" y="3" z="0"/>
    </entity>
</entities>
<!--below is the relationships among metaballs-->
<relationships>
    <relationship from="0" to="1" weight="1"/>
    <relationship from="0" to="2" weight="0.5"/>
    <relationship from="0" to="3" weight="1"/>
    <relationship from="0" to="4" weight="0.5"/>
    <relationship from="0" to="8" weight="0.5"/>
    <relationship from="1" to="5"       weight="0.25"/>
    <relationship from="1" to="6" weight="0.5"/>
    <relationship from="6" to="7" weight="0.25"/>

</relationships>
<layout estimationValue="32" lineCrossing="0" lineObjectCrossing="0"
totalLength="32"/>
</entity_relation>
```

Appendix B: Example of Execution Trace for Dynamic Diagrams of User

Execution Number	Class Number	Line Number	Execution Number	Class Number	Line Number
1	Elevator	110	61	Elevator	116
2	Elevator	111	62	Elevator	117
3	Elevator	112	63	Elevator	118
4	Elevator	17	64	Elevator	70
5	Elevator	18	65	Elevator	71
6	Elevator	19	66	Elevator	72
7	Panel	9	67	Elevator	74
8	Panel	10	68	Elevator	75
9	Panel	11	69	Elevator	77
10	Panel	12	70	Elevator	78
11	Panel	13	71	Elevator	80
12	Button	14	72	Elevator	83
13	Button	15	73	Elevator	87
14	Panel	13	74	Elevator	88
15	Button	14	75	Panel	38
16	Button	15	76	Panel	39
17	Panel	13	77	Button	18
18	Button	14	78	Button	19
19	Button	15	79	Elevator	103
20	Panel	13	80	Elevator	78
21	Button	14	81	Elevator	80
22	Button	15	82	Elevator	83
23	Panel	13	83	Elevator	87
24	Button	14	84	Elevator	90
25	Button	15	85	Elevator	91
26	Panel	13	86	Panel	21
27	Button	14	87	Panel	22
28	Button	15	88	Panel	23
29	Panel	13	89	Panel	24
30	Button	14	90	Button	22
31	Button	15	91	Button	23
32	Panel	13	92	Button	26
33	Button	14	93	Panel	24
34	Button	15	94	Button	22
35	Panel	13	95	Button	23
36	Button	14	96	Button	26
37	Button	15	97	Panel	24
38	Panel	13	98	Button	22
39	Button	14	99	Button	23
40	Button	15	100	Button	26
41	Panel	13	101	Panel	24
42	Button	14	102	Button	22
43	Button	15	103	Button	23
44	Panel	13	104	Button	26
45	Button	14	105	Panel	24
46	Button	15	106	Button	22
47	Panel	13	107	Button	23

48	Button	14	108	Button	26
49	Button	15	109	Panel	24
50	Panel	13	110	Button	22
51	Button	14	111	Button	23
52	Button	15	112	Button	26
53	Panel	13	113	Panel	24
54	Button	14	114	Button	22
55	Button	15	115	Button	23
56	Elevator	20	116	Button	26
57	Elevator	21	117	Panel	24
58	Elevator	22	118	Button	22
59	Elevator	114	119	Button	23
60	Elevator	115	120	Button	26
…	…	…	…	…	…

Appendix C: **Software Usability Measurement Inventory (SUMI)**

Your name

Name of software

Date

NB *the information you provide is kept completely confidential, and no information is stored on computer media that could identify you as a person.*

This questionnaire has fifty statements. Please answer them all. After each statement there are three boxes.

You should check the first box if you generally AGREE with the statement. Check the middle box if you are UNDECIDED, or if the statement has no relevance to your software or to your situation. Check the right box if you generally DISAGREE with the statement.

In checking the left or right box you are not necessarily indicating *strong* agreement or disagreement but just your general feeling most of the time.

AGREE UNDECIDED DISAGREE

Put a x in the box of your choice

		Agree	Undecided	Disagree
1	This software responds too slowly to input.	☐	☐	☐
2	I would recommend this software to my colleagues.	☐	☐	☐
3	The instructions and prompts are helpful.	☐	☐	☐
4	The software has stopped unexpectedly sometimes.	☐	☐	☐
5	Learning to use this software is difficult to start with.	☐	☐	☐
6	Sometimes, I don't know what I should do next with this software.	☐	☐	☐
7	I enjoy the time I spend using this software.	☐	☐	☐
8	I find the help information given by this software is not very useful.	☐	☐	☐
9	If this software stops, it is not easy to restart.	☐	☐	☐
10	It takes too long to learn what to do with this software.	☐	☐	☐
11	I sometimes wonder if I'm using the right command.	☐	☐	☐
12	Working with this software is satisfying.	☐	☐	☐
13	System information is presented in a clear and understandable way.	☐	☐	☐
14	I feel safer if I use only a few familiar commands or operations.	☐	☐	☐
15	The software documentation is very informative.	☐	☐	☐
16	This software disrupts the way I normally like to arrange my work.	☐	☐	☐
17	Working with this software is mentally stimulating.	☐	☐	☐
18	There is never enough information on the screen when I need it.	☐	☐	☐
19	I feel in control of this software when I am using it.	☐	☐	☐

20	1 prefer to stick to the functions that I know best.	☐	☐	☐
21	This software is inconsistent.	☐	☐	☐
22	I wouldn't like to use this software every day.	☐	☐	☐
23	I can understand and act on the information provided by this software.	☐	☐	☐
24	It's hard to do non-standard things with this software.	☐	☐	☐
25	There is too much to read before you can start using the software.	☐	☐	☐
26	Getting your tasks done with this software is easy.	☐	☐	☐
27	Using this software is frustrating.	☐	☐	☐
28	The software has helped me overcome any problems I've had in using it.	☐	☐	☐
29	The speed of this software is about right.	☐	☐	☐
30	I often have to go back and look at the user manuals.	☐	☐	☐
31	It's clear that user needs have been taken into consideration.	☐	☐	☐
32	Using this software sometimes makes me feel tense.	☐	☐	☐
33	The organisation of the menus or information lists seems fairly logical.	☐	☐	☐
34	You don't have to do much typing with this software.	☐	☐	☐
35	It's hard to learn new functions.	☐	☐	☐
36	You have to go through too many steps to get something to work.	☐	☐	☐
37	Sometimes, this software gives me a headache!	☐	☐	☐
38	Error prevention messages are inadequate.	☐	☐	☐
39	It's easy to make the software do exactly what you want.	☐	☐	☐
40	I'll never learn all the functions in this software product.	☐	☐	☐
41	Sometimes the software doesn't do what I expect it to do.	☐	☐	☐
42	The software has a very attractive presentation.	☐	☐	☐
43	The amount or quality of the help information varies across the system.	☐	☐	☐
44	It's relatively easy to move from one part of a task to another.	☐	☐	☐

45	It's easy to forget how to do things with this software.	☐	☐	☐
46	Sometimes this software behaves in a way I don't understand.	☐	☐	☐
47	This software is very awkward to use.	☐	☐	☐
48	You can see at a glance what the options are at each stage.	☐	☐	☐
49	Getting data files in and out of the system is not easy.	☐	☐	☐
50	I have to seek assistance when I use this software.	☐	☐	☐

Please be sure to check each item.

Thank you. [51]