Software Visualization

PeiLing Li

A Major Report

In

The Department

Of

Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montreal, Quebec, Canada

March 2003

Abstract

Software Visualization

Peiling Li

Software visualization is a significant force in software engineering. As the sizes of software systems are becoming larger and more complex, program comprehension is becoming more difficult. The tasks of program comprehension involve implementation, maintenance, testing, debugging, mental model construction and verification. Software visualization is one promised way to support the tasks of program comprehension. Through software visualization, graphics and animations are built to help illustrate and present the computer program. In recent years, many related visualization techniques have been developed and used to build visualization tools. The major goal of visualization tools is to support program comprehension.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. Introduction

Software visualization is a significant force in software engineering. As the sizes of software systems are becoming larger and more complex, program comprehension is becoming more difficult. The tasks of programming, understanding, and maintaining are becoming more and more difficult because the size of software systems is increasingly larger and complex. Software visualization (SV) is one promised way to support these tasks. Software visualization consists of the use of computer graphical artifacts and animation to help illustrate and present computer programs, processes, and algorithms. Roman and Cox define *program visualization* as the mapping from programs to graphical representations [RC92]. Software visualization is necessary because software is a textual expression; it seems the only way to have a global view of the system by reading the source code. When the information amount is large, the reader is overwhelmed with information. Graphical representations have been recognized as having an important impact in communicating from the perspective of both writers and readers [LW86]. The use of software visualization for program understanding or comprehension is the act of perceiving the meaning and structure of a program. Additionally, program comprehension is also useful in program debugging and testing. Programmers must understand programs to devise complete and comprehensive test cases and to locate bugs. The need for tools to support program comprehension is well documented [Oman90]. Visualization tools should support the user in the performance of program comprehension tasks during implementation, maintenance, testing and debugging, to aid the user in the construction of a mental model, and the performance of program strategies.

The remainder of this paper will be organized as follows: Section 2 discusses the background of visualization and program comprehension, Section 3 discusses software visualization and related techniques, Section 4 summaries visualization tools, Section 5 presents a summary.

2. The background of visualization and program comprehension

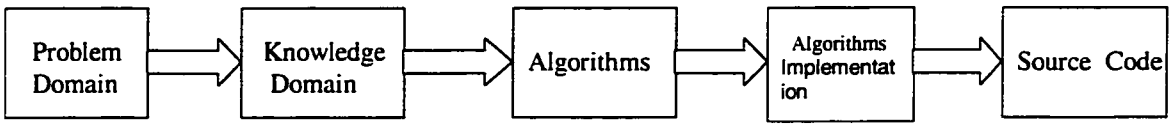
Software engineering is concerned with improving the productivity of the software development process and the quality of the systems it produces. However, as currently practiced, the majority of the software development effort is spent in maintaining existing systems rather than developing new ones [Rug81]. Estimates of the proportion of the resources and time devoted to maintenance range from 50% to 75%[Boe81, ST78]. Maintenance can be defined as the managing processes of system change [Boe81, ST78]. Maintenance is difficult and expensive for various reasons, such as, the original source code writer leaving, documentation being lost, out of date, or incomplete. In fact, maintenance occurs over the whole life cycle of software. The activities of maintenance include fixing problems, adapting to the new environment, adding new features. However, it is only in recent years that work on program comprehension has become an independent discipline closely linked to the field of software maintenance. Program comprehension activities have a destructive effect on the productivity of maintenance programmers. Comprehension is estimated at 50% - 60% of the maintenance effort. In fact available estimates indicate the percentage of maintenance time consumed on program comprehension ranges from 50% up to 90%[Cor89, Sta84, LS94]. Hence, work on program comprehension presents a tremendous potential for improvement in the productive of maintenance programmers and reduction of overall software life cycle cost.

2.1 Program Comprehension

The field of software comprehension is summarized in Corbi90, [RBCM91]. Software comprehension involves both software engineering and cognitive science. Rugaber

[Rug81] defined program comprehension as t*he process of acquiring knowledge about a computer program.* Programmers use programming knowledge, domain knowledge, and comprehension strategies when trying to understand a program. For example, one might extract syntactic knowledge from the source code and rely on programming knowledge to form semantic abstraction [MWT94]. Shneiderman [RC92], defines syntactic and semantic program knowledge. Syntactic knowledge is language dependent and concerned with the statements and basic units in a program. Semantic knowledge is language independent and is built in progressive layers until a mental model is acquired through the chunking and aggregation of other semantic components and syntactic fragments of text.

Brooks [Bro83] defined *comprehension as the reconstruction of the domain knowledge used by the initial developer.* Domain knowledge is knowledge about a particular domain such as operating systems or UNIX systems. In this theory, understanding proceeds by recreating the mappings from the problem domain to the programming domain through intermediate domains. The problem domain or application domain consists of problems in the real world. Hence, comprehending a program requires recreating the mapping between domains. Figure 1 and figure 2, taken from [Cru01], show the steps involved to move from problem domain to application domain and some gaps between them.



Step1 Step 2 Step 3 Step 4

Figure 1. Program Creation

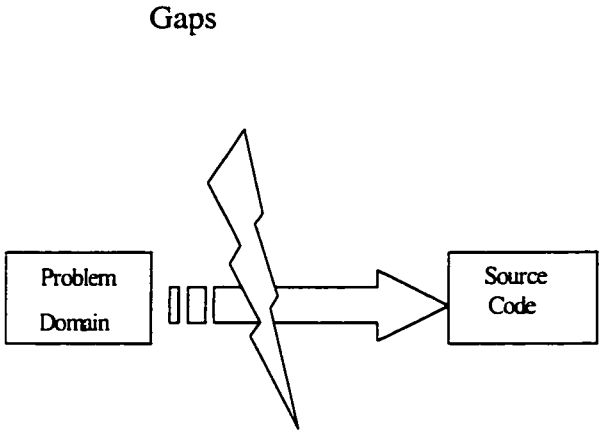


Figure 2. Program Comprehension

In order to map the different domains, program comprehension has to bridge different concepts via gaps. The following five gaps were indicated in [Rug81]:

- The gap between a problem from some application domain and a solution to it in some programming language.

- The gap between the concrete world of physical machines and computer programs and the abstract world of high-level design descriptions.
- The gap between the desired coherent and highly structured description of a system as originally envisioned by its designers and the actual system whose structure may have disintegrated over time.
- The gap between the hierarchical world of programs and the associational nature of human cognition.
- The gap between the bottom-up analysis of the source code and the top-down synthesis of the description of the application.

In order to understand the process of program comprehension, cognitive models were have been proposed in the literature.

2.2 Cognitive models

Cognitive models reference both existing and newly acquired knowledge to build a *mental model* [MV94]. A *mental model* is a set of beliefs that you hold about how a piece of software, or a software feature, works. A cognitive model describes the cognitive processes and information structures used to form a mental model [MV95]. In the literature, there are four accepted categories of theories that describe the cognitive processes involved in program comprehension: bottom-up program comprehension, top-down program comprehension, opportunistic program comprehension, and an integrated meta-model of program comprehension [SFM99].

2.2.1 Bottom-up program comprehension

Bottom-up comprehension permits low-level code to be generated first, in an attempt to build up to the goal [Shn80]. Bottom-up theories are based on the notion that a programmer understands a program by iteratively abstracting and connecting together 'chunks' of code. Chunks are pieces of code that have own meaning. The strategy of chunking is, then the smallest chunks, the bigger, etc., until the whole program is understood.

2.2.1.1 Shneiderman's Model

Shneiderman proposed that programs are understood bottom-up, by reading source code and then mentally chunking low-level software artifacts into meaningful, high-level abstractions [Shn80]. These abstractions are further grouped until a high-level understanding of the program is formed. The Shneiderman comprehension model is shown in figure 3 [SM79].

Shneiderman's view of the comprehension process consists of three levels:

- Low-level: Comprehension of the function of each line of code.
- Mid-Level: Comprehension of the nature of the algorithms and data, and high-level: comprehension of overall program function. It is possible to understand each line of code and not to understand the overall program function. It is also possible to understand the overall program function and not to understand the individual lines of code, not the algorithms and data. Mid-level comprehension involves knowledge of the control structures, module design, and data structures,

which can be understood without knowledge at the other two levels. Thorough comprehension involves all three levels of understanding.

- High-Level: The programmer's semantic and syntactic knowledge can help programmers to get a high level abstraction of a given program, finally an internal semantic form is built. At the highest level, the programmers know what program does and how it does, even the low-level structures such as algorithms or data structure. The internal semantic form represents the understanding of the program; it is independent of and can be expressed in other languages or context.

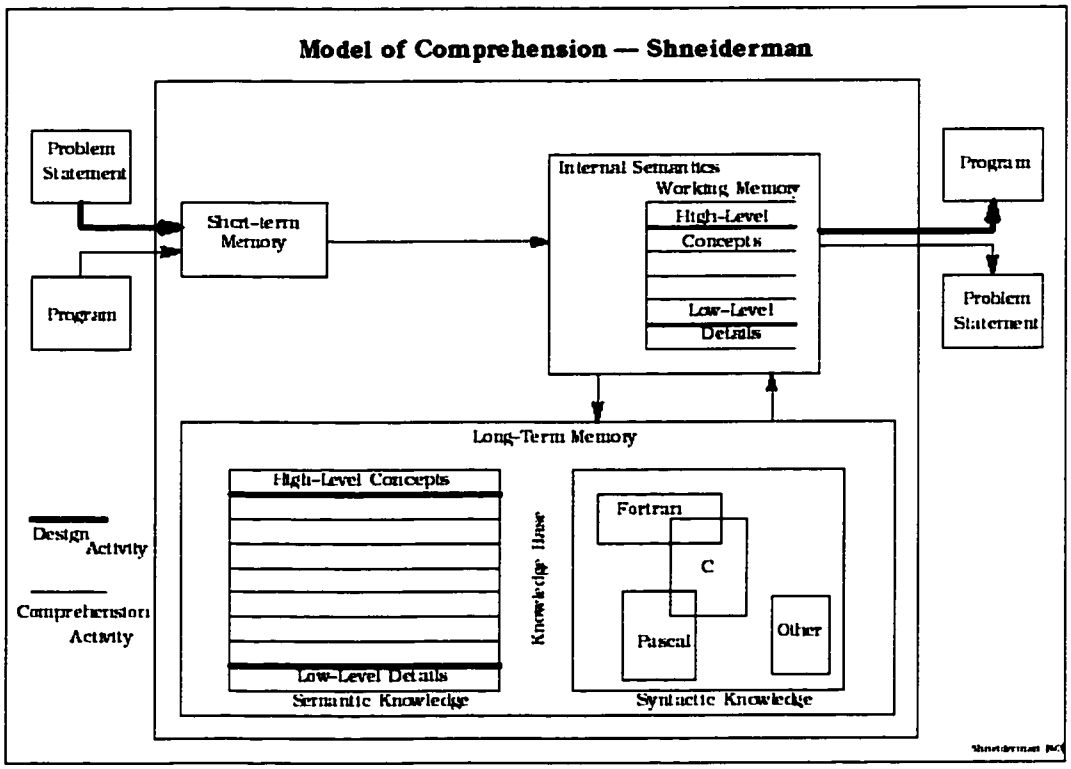


Figure 3. Shneiderman's Model

2.2.1.2 Pennington's Model

Pennington [Pen87] also suggests that program comprehension is bottom-up. When the source code is totally new to the programmer, a mental model called *program model* is built first. A program model is a control flow abstraction that captures the sequential behaviors of the program execution. Pennington uses *text structure* and *programming plan knowledge* to explain the development of a program model. *The text structure knowledge consists of the control primes used to build the program model.*

Programming plan knowledge, consisting of programming concepts, is used to exploit existing knowledge during the comprehension task and to infer new plans for storage in long-term memory. Chunking is the main activity in the comprehension process. When more and new abstract program knowledge is then built by chunking code structures into more abstract structures, a program model is built. After the program model, a situation model is mentally developed. The development of a situation model requires the knowledge of real-world domain, using program model to create a data-flow/functional abstraction. The mechanism used for situation model is *domain plan knowledge. Domain plan knowledge* is used to derive a mental representation of the code in terms of real-world objects, organized as a functional hierarchy in the problem domain language. A situation model encompasses chunked plan knowledge, and the program model consists of a hierarchy-chunked components. Figure 4 [Pen87] is a graphical representation of Pennington's model, The right half illustrate the process of program model building, and the left half describes the situation model construction.

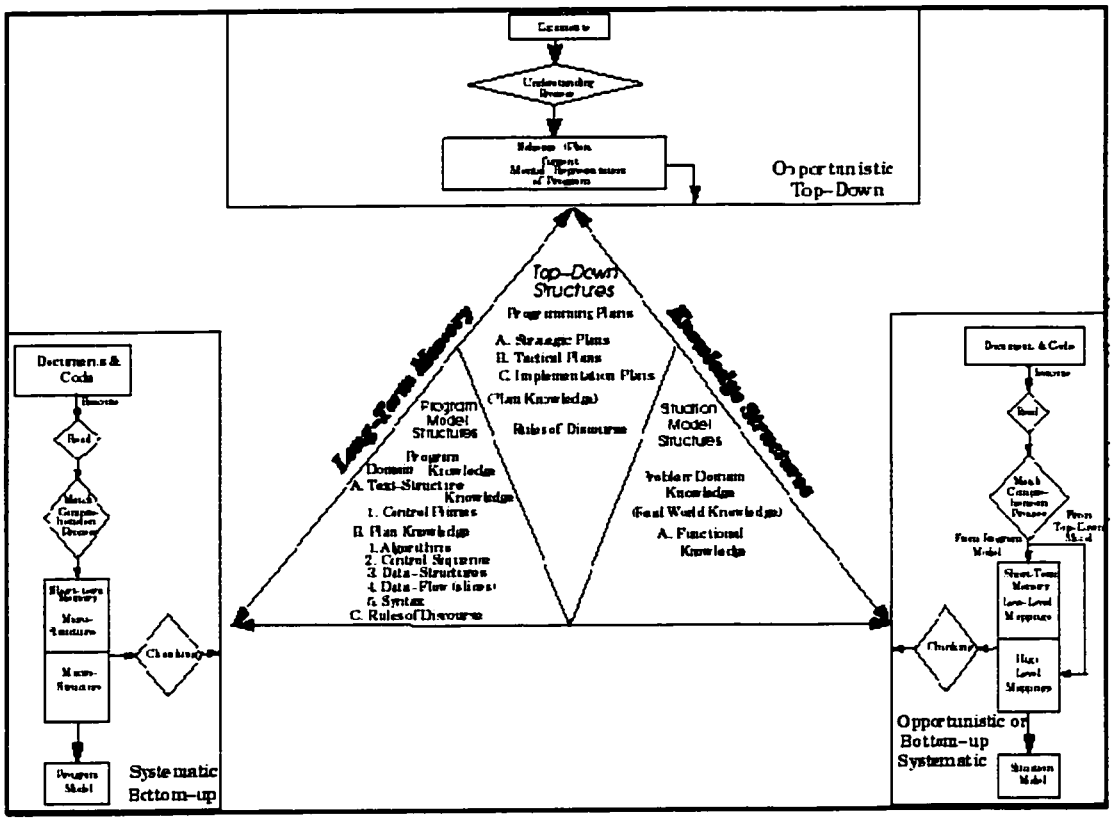


Figure 4. Penning's Model

2.2.1.3 Comparision of Shneiderman's model and Penning's model

The cognitive structures of Shneiderman's model are multileveled. Information from the outside world , to which the programmer pays attention, such as descriptions of the to-be-programmed problem, enter the cognitive system into *short-term memory,* a memory store with a relatively limited capacity (Miller,1956, suggests about seven chunks), and

which performs little analysis on the input information. A *short-term memory*'s capacity is limited. The programmer's permanent knowledge is stored as *long-term memory*, with unlimited capacity for organized information. Information from *short-term memory* and existing concepts from *long-term memory* are integrated in a *working memory* [Feigenbaum, 1974] which represents a store that is more permanent that short-term but less permanent than long-term memory, and in which information from short-term and long-term memory may be integrated into new structures. The result is used to generate a solution, or as learning, is stored in long-term memory for future use. Thus, Shneiderman's model has a hierarchical organization of knowledge and separates it as semantic and syntactic knowledge. Shneiderman's model focuses on the form of mental representation, but it lacks on the details of knowledge construction. For example, there is no mechanism for abstraction.

On the contrary, Pennington's model is more detailed and includes the implementation mechanisms of cognitive process. The program model's mechanisms are *text structure* and *programming plan knowledge*, and situation's mechanisms are cross-reference and chunking. The major drawback of Pennington's model is the lack of higher level knowledge structures such as design or application domain knowledge.

2.2.2 Top-down program comprehension

The theory of top-down comprehension is that the comprehension tasks start by gaining an understanding of the overall goals of the program; each subtask is then viewed from the perspective of how it relates to that goal. The programmer uses their own experience and attempts to confirm their expectations.

2.2.2.1 Brooks' Model

Ruven Brooks' model deals with the comprehension of completed programs [Bro77, Bro83]. A completed program means that the source code has been implemented. Brooks' model has its basis in areas outside of computer science, such as thermodynamics problem solving, physics problem solving, and chess. The model was initially created to explain four major source of variation observed in the act of program comprehension.

1) The functionality of the program to be understood. Why do programs that perform different computations vary in comprehensibility?

2) The differences in the program text. Why do programs that are written in different languages differ in comprehensibility, even though the same calculation is performed in each?

3) The motivation the understander has in comprehending the program. Why does the comprehension process vary depending on whether the motivation is to debug the program versus enhancing the program?

4) Individual differences between people ability's in comprehending a program's purpose. Why does one person find a program easier to comprehend than does another?

To account for these four areas of variation, Brooks created a model based on three main ideas. There three ideas are:

1) The programming process is the construction of mappings from a *task* domain, through one or more *intermediate* domains, to the *programming* domains.

2) The comprehension process of that program is the reconstruction of all or part of those mappings.

3) The reconstruction process is expectation-driven by the creation, confirmation and refinement of hypotheses. These hypotheses describe the various domains, and the relationships between them.

Brooks [Bro83] suggests that the central strategy employed in top-down program comprehension is hypothesis formation and evaluation. He starts from the assumption that in the development of a program a programmer creates a mapping between the application domain and the domain of programming. Comprehension involves the reconstruction of this mapping, through several intermediate domains. This is an iterative, hypothesis-driven process that begins with the formulation of a primary hypothesis that expresses a global description of the program goal. Next, subsidiary hypotheses are formulated to refine the primary hypothesis in a hierarchical faction. Hypotheses are iteratively refined, passing through several knowledge domains, until they can be matched to specific code in the program or some related document. For example, a hypothesis may state that a particular equation (math domain) expresses cost (accounting domain). A procedure name *FCFS* may generate the hypothesis that a first-come-first serve algorithm is used for process scheduling.

Figure 5[4] illustrates Brooks' model. Knowledge shown as triangles can be used directly for hypothesis generation in the mental model or it can be matched from one domain into another.

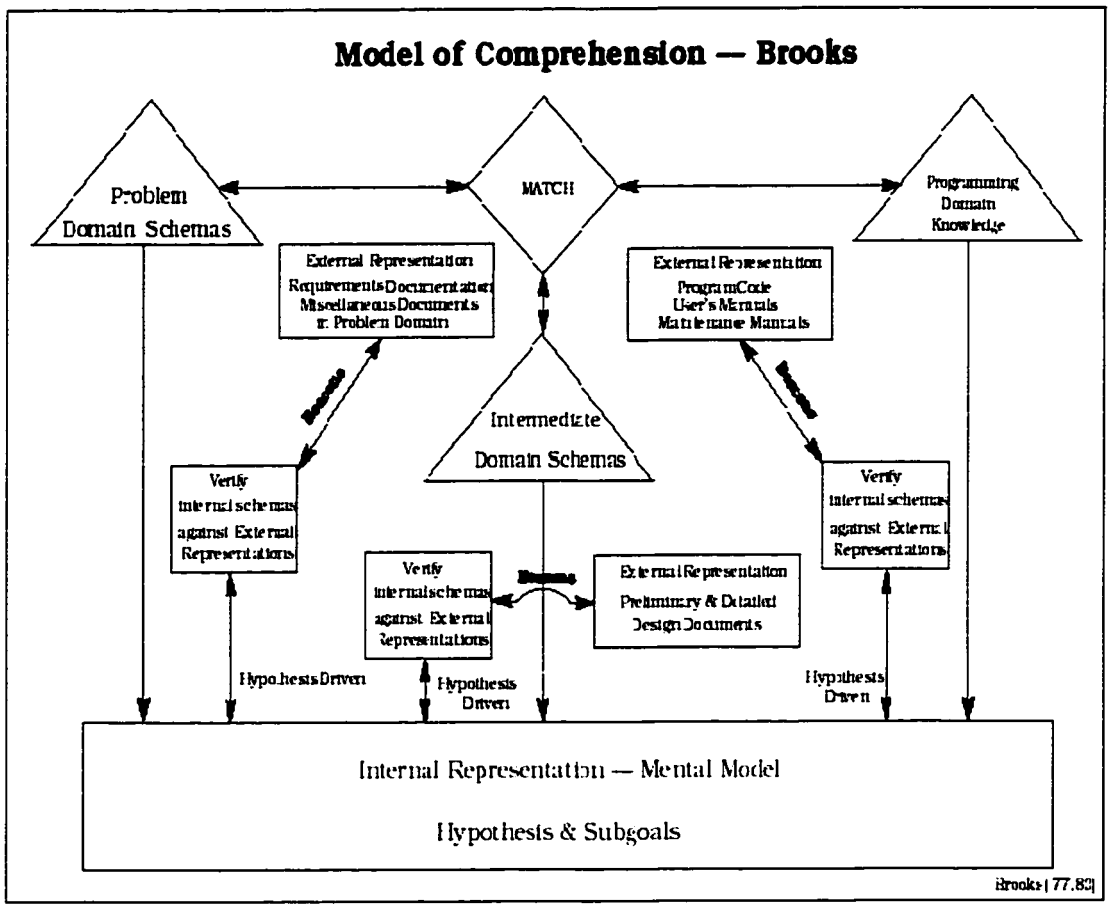


Figure 5. Brooks' comprehension model

2.2.2.2 Soloway and Ehrlich's Model

The Top Down program understanding model [SE84, SAE88] typically applies when the code or type of code is familiar. In this case, the code can be decomposed into a hierarchy of elements typical of that program type. For example, an expert of operating systems can easily decompose a new operating system into standard components, such as a process scheduler, a virtual memory manager, and a file system. The mental model constructed in this way consists of a hierarchy of goals and plans. Rules of discourse capture

conventions of programming, such as algorithm and data-structure implementations and coding standards, and are used to decompose goals into plans and plans into lower level plans. Goals denote intentions, and plans denote techniques for realizing intensions. Plans work as rewrite rules that convert goals into subgoals and finally into program code. Program comprehension is defined as the process of recognizing plans in code, combining these plans by reversing the rewrite rules to form subgoals, and combining the subgoals into higher-level goals. The model includes three different types of plans: *strategic plans*, *tactical plan*, and *implementation plans*. *Strategic plans* describe goal strategy enacted in the program to accomplish its goal. *Tactical plans* are local strategies for solving a problem. *Implementation plans* describe the characteristic of the language and are used to implement tactical plans.

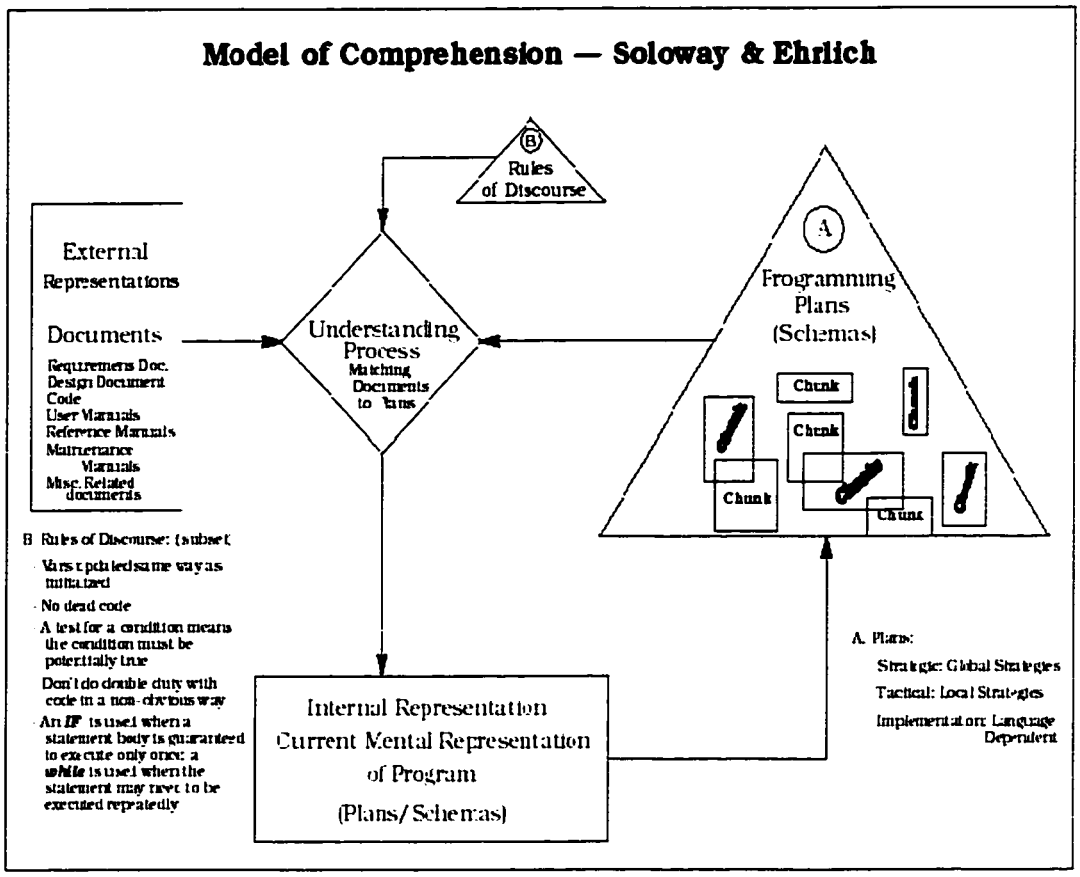


Figure 6. Soloway & Ehrlich's Model

Figure 6, taken from [MV94] shows the model's three major components: 1) The triangles represent knowledge (programming plans or rules of discourse). 2) The diamond represents the understanding process. 3) The rectangles illustrate internal representations of the program.

2.2.2.3 Comparison of Brooks' Model and Soloway and Ehrlich's Model

Brooks' model emphasizes the mappings between domains and their interrelations. All levels of domain eventually meet the " actual code" domain level; it is justified that a

larger effort be made in supporting this bottom level domain and the relationships. One way to do this is to make the binding stage more explicit by showing the relationships between an intermediate hypothesis and the code to which it has been bound. This could also involve some sort of status to indicate if a given section of code has been bound at all, or possibly even that it has been bound twice. Another issue is that the mental model is constructed in one direction only, from the problem domain to the program domain. The knowledge so structured is undefined. It would not be possible to switch from one level abstraction to another going in an opposite direction.

A Knowledge base is required in Soloway's model. The element of domain knowledge is similar to Brooks' domains. The highest-level abstractions in the mental model are emphasized. But the relationships between high level domains (e.g. "invoicing" to functional decomposition") are not addressed by plans. This is a direct consequence of the formal, rigid structure of plans: they have gained expressive power at the low level domains by sacrificing the power needed to express high-level domain relationships.

2.2.3 Opportunistic Program Comprehension

Opportunistic theories are a mixture of the bottom-up and top-down processes where the programmer uses an as needed, rather than a systematic, approach to understand the actual code [MV93c]. Opportunistic comprehension is characteristic of experienced programmers. Who apply their knowledge to formulate hypotheses, where possible, and analyze code to identify chunks and other high level structures when necessary.

2.2.3.1 Letovsky's Model

Letovsky [Let86] views programmers as opportunistic processors capable of exploiting either bottom or top-down cues. There are three components to his model, the knowledge base, mental model, and assimilation process. Knowledge base encode the programmer's expertise and background knowledge. Mental model encodes the programmer's current understanding of the program. Assimilation process describes how a mental model evolves using the programmer's knowledge base and program source code and documentation. The mental model is organized into three different layers: the specification layer, which describes the program goals, the implementation layer, which expresses the lowest level abstraction, and the annotation layer, which links each goal in the specification layer with its relationship in the implementation layer. Assimilation is opportunistic and occurs either top-down or bottom-up depending on the programmer's initial knowledge base. Programmer's *Inquiry episodes* are a key part of the assimilation process. Such an episode consists of a programmer asking a question, conjecturing an answer, and then searching through the code and documentation to verify or reject the conjecture. Fiugre 7 [MV94] shows Letovsky's Model.

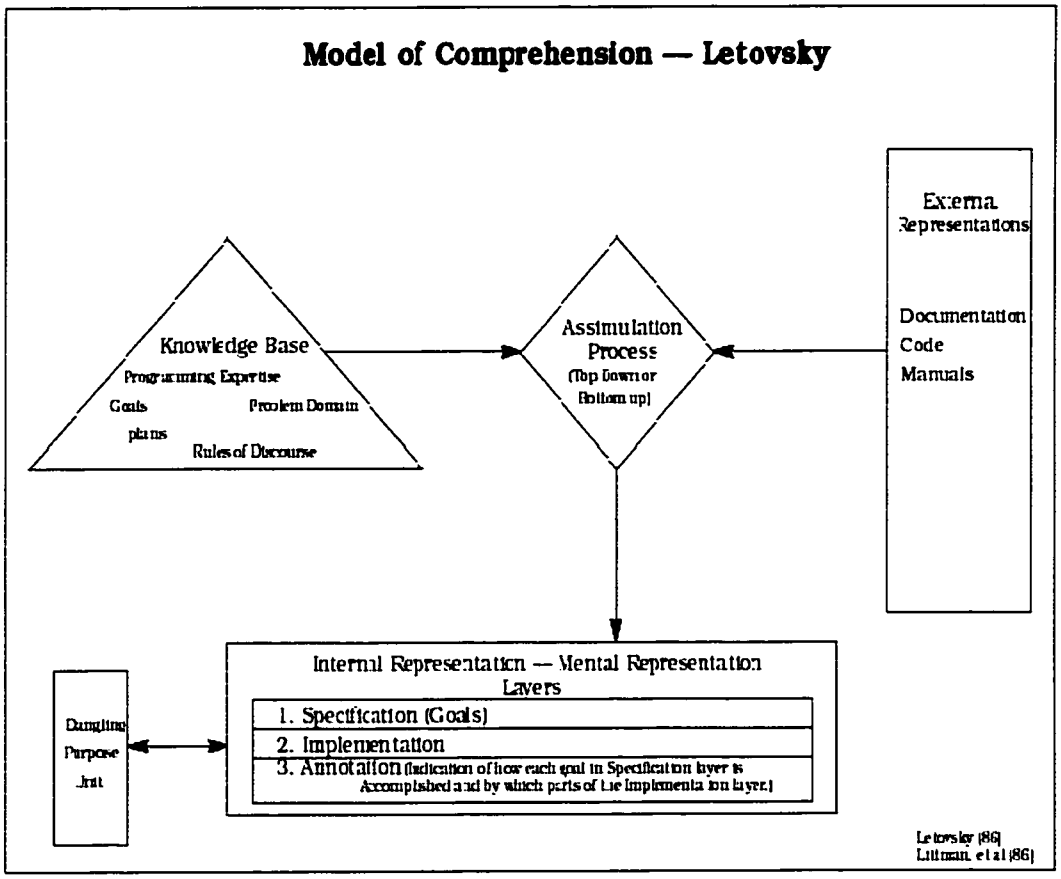


Figure 7. Letovsky's Model

2.2.3.2 Comparison of Letovsky's Model

Letovsky's model produces either top-down or bottom-up comprehension depending on the program's initial knowledge base; it is suitable for experienced programmers. Letovysky's model is the most general cognition model. The mental representation is described in detail, but there is a lack of description of how the knowledge assimilation process works or how knowledge is incorporated into the mental model representation beyond the statements that it occurs.

2.2.4 An Integrated Meta-Model of Program Comprehension

Von Mayrhauser and Vans [MV93a] combined the top-down, bottom, and opportunistic approaches into a single metamodel. They propose an integrated cognition model that combines features of several existing models, primarily Soloway and Ehrlich's top-down model [SE84, SAE88] and Pennington's program and situation models [Pen87]. Understanding is built concurrently at several levels of abstractions by freely switching between the three comprehension strategies.

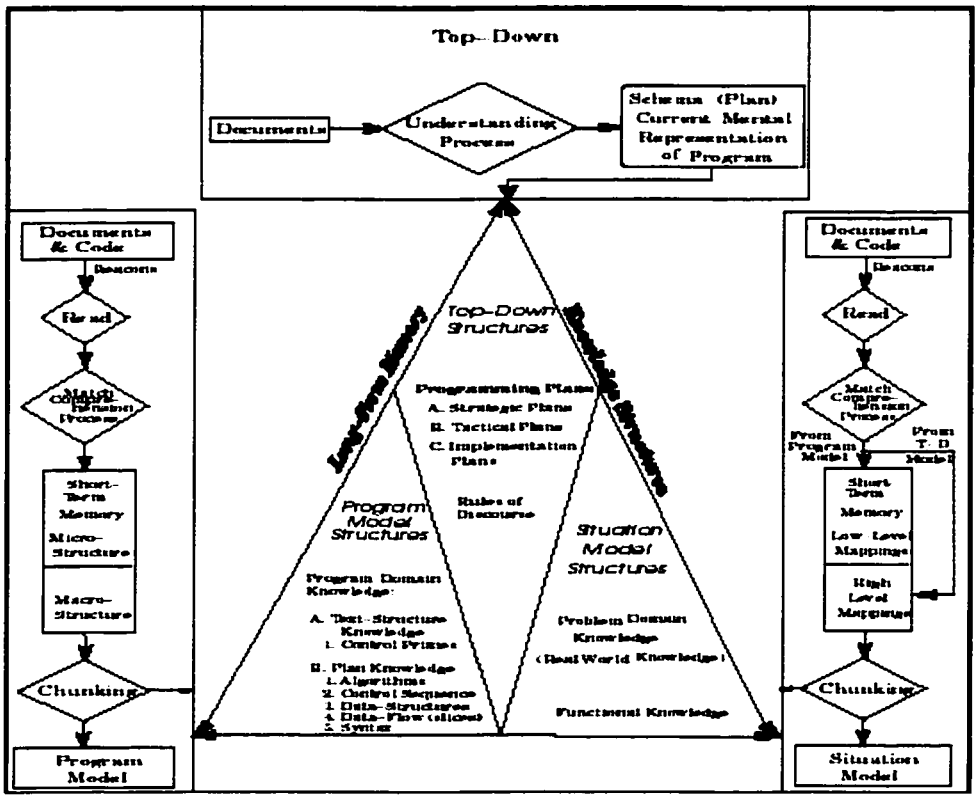


Figure 8.Integrated Code Comprehension Meta-Model

The integrated code comprehension meta-model consists of four main components: a program model, a situation model, a top-down model, and a knowledge base. The first three components reflect mental representations of comprehension and the strategies used to construct them. The fourth component refers to knowledge needed to perform the process of comprehension. Each model component represents both the internal representation of the code and the strategy to build this internal representation. The knowledge base furnishes previously acquired information related to the comprehension task. During comprehension, new knowledge is developed and stored into knowledge base for future use. The top-down model of the integrated model is typical active if the code or type of code is familiar within its application domain. If the programmers are unfamiliar with the code, the programmers will switch to the bottom-up model. The main feature of the integrated meta-model is that any of the three model components may be in effect during the understanding process to accomplish a comprehension goal [MV93b, MV93c].

Figure 8, taken from [MV93a], shows that each model component has its own preferred types of knowledge. [MV93b, MV93c] contain thorough discussions of the integrated meta-model and its component models.

2.2.4.1 Comparison of the Integrated Meta-Model

The integrated model combines the top-down understanding of [SE84, SAE88] with the bottom-up understanding of [Pen87], opportunistic approach [Let86], recognizing that for large systems a combination of approaches to understanding becomes necessary [MV93a]. Experiments showed that programmers switch between all three of these

comprehension models [MV93b, MV93c]. For example, during program model construction a programmer may choose top-down when he recognizes a beacon indicating a common task such as sorting. This leads to the hypothesis that the code sorts something. A sub-goal is generated and the code for clues is searched and selected to accomplish the sub-goal. If during the code searching, the programmer finds some codes he doesn't familiar with, he may come back to bottom –up. The structures of integrated Meta model built by any of the three model components are accessible by any others. However, each model component has its own features of knowledge. Compared to the opportunistic approach, the integrated comprehension promises the understanding is developed at several levels of abstractions by freely switching between the three comprehension strategies, and opportunistic approach chooses either bottom-up or top-down comprehension at the beginning depending on the programmer's initiate knowledge.

3. Software Visualization and Related Techniques

Program visualization is simply defined as the use of graphic artifacts to enhance the understanding of programs [EO94, PBS93, RC92]. In a wider context, it is sometimes also called *software visualization* [PB93]. Through software visualization, graphical icons help building program understanding. Software visualization can be done at different levels, such as at the abstract algorithm level, or the language level. For example, SHriMP (Simple Hierarchical Multi-Perspective) supports for seamless exploring software structure and browsing source code, with a focus on effectively assisting hybrid program comprehension strategies [SWFM97]. SHriMP visualization technique integrated the visualization techniques such as a nested graph, fisheye views or focus + context, pan + zoom, hypertext links, animation etc. SHriMP can be used to visualize source code of a complex software system. SHriMP can be combined with an ontology editor such as Jambalaya for knowledge acquisition.

3.1 Reverse Engineering

Software systems that are developed specially for an organization have a long lifetime. Some software systems developed many years ago are still in used using technologies that are now obsolete. They are known as legacy systems [CC90] and have become business-critical for many companies. These legacy systems need to be maintained and evolved due to many factors, including error correction, requirements change, business rules change, structure re-organization, etc. Legacy systems are difficult to understand and are maintained because of their size and complexity as well as the history of their evolution. Reverse engineering is one of promising approaches in program

understanding. *Reverse engineering* is the process of analyzing a subject system to identify the system components and their interrelationships, and create representations of the system in another form or in a high level abstraction [MWT94]. The goal of reverse engineering is to extract information from the exiting software systems to better understand them [LCCCLY95]. The information includes the underlying features of a system, such as [Con87]:

- 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.

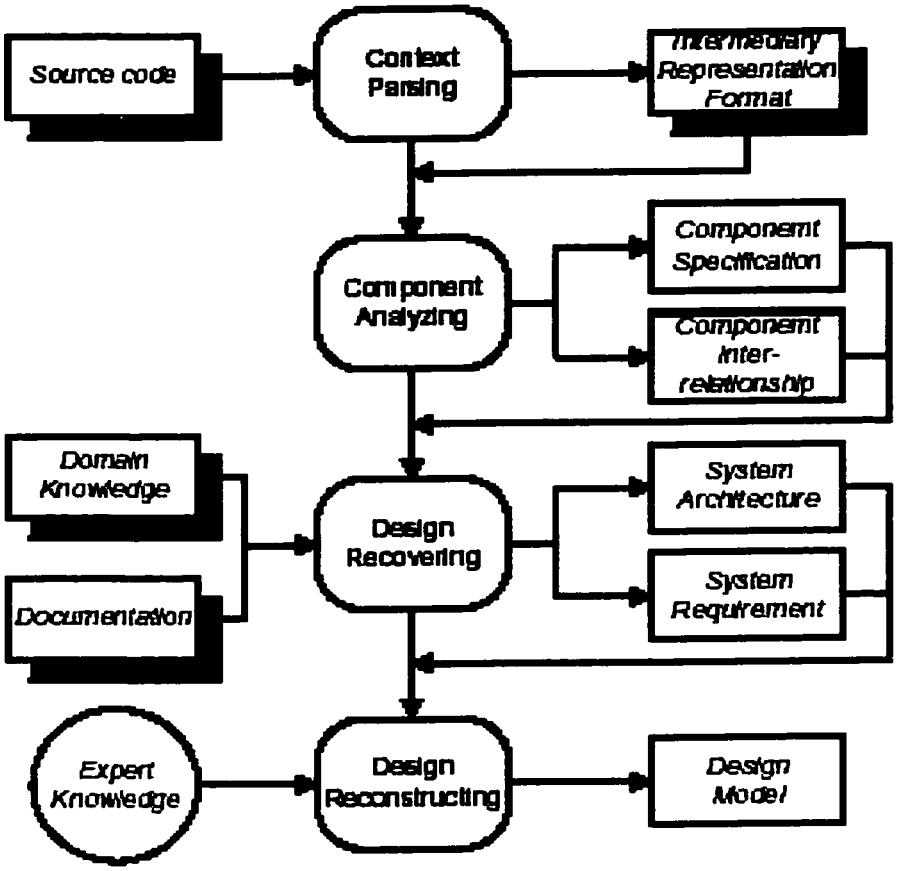


Figure 9. Reverse Engineering Process

The process of reverse engineering can be divided into four phases: Context Parsing, Component Analyzing, Design Recovering and Design Reconstructing, as shown in Figure 9 [Con87].

- Context Parsing Phase - Analyzing source code and extracting information from the source code. An intermediate representation is created, such as the Abstract Syntax Tree (AST), which can be accepted as input of the next phase.
- Component Analyzing Phase – The component's artifacts are revealed. The artifacts include: structure chart, variables' attributes, functions' information,

program slices, call graphs, data flow, definition- use graph and control dependencies.

- Design Recovering Phase - To extract original requirements and/or design knowledge from source code. a high-level view is obtained.
- Design Reconstructing Phase – A design model is generated from the last phase. The design model offers not only functionality and system behavior, but also the correct architecture.

During the reverse engineering process, the source code is not altered, although additional information about it is generated [LCCCLY95].

3.2 Information Visualization Techniques

Information visualization, as opposed to scientific visualization, aims to visualize abstract data that may have no natural visual representation. This data can be very complex, containing a great number of elements structured hierarchically, in a network, linearly, or even lacking structure. The use of information visualization, rather than using the raw data, is done for several reasons, some of which are: [Knight98]:

- Being able to display a large amount of information in one view and thus provide an overview.
- Being able to see correlations or patterns that may have otherwise been missed had only the figures been used.
- Trying to display structural relationships and context that may be more difficult to detect by individual retrieval requests (provided by Card et al.[CKM91]).

- Providing an effective way of going between overview abstractions and detail of the data.

Numerous methods have been studied for graphically representing information within a limited amount of screen space. There are several considerations that have be made in choosing a particular technique [LM94]. The method must be suitable for the information that you are going to display- some are dedicated to certain styles of information. In any system that is required to support real time user interaction, good performance is imperative. Other implementation issues must be considered, including the hardware requirements (e.g., memory consumption, 3D acceleration), and the screen resolution available to the interface. In the following section of this paper, I present several implementation techniques which are hypertext, focus + context, shading/color/texture, dimension 2D versus 3D, and animation.

3.2.1 Hypertext

Since the introduction of the World Wide Web, hypertext has become a very popular and familiar technology. The basic tools such as the html language, browser, Java applets etc. are widely available and understood by many programmers. The same technology can be used in information visualization. Hypertext is a method of document navigation that facilitates non-sequential document reading [Nie90, Nie95]. Traditional documents are typically designed to be read from beginning to end in a sequential manner. In particular, physical paper documents must be presented in this way (page 1 precedes page 2 which precedes 3..). However, many paper documents are not intended to be read sequentially (e.g., reference manuals, dictionaries). Non-sequential navigation including the content list, index, page reference, and footnote. Although these techniques are effective, the

reader has to flip the pages to navigate. In a hypertext document, many sequential restrictions can be lifted and new navigation methods can be implemented. Components may act as links to other pages of the document. A link (also called an anchor) could be a word, words or a figure. When a link is activated, the reader directly navigates to the point in the hypertext that the link indicates. In a hypertext version of the document, each word in the document glossary can be linked to each occurrence of that word in the document. By selecting the obscure word in the text, the reader navigates the link directly to the word's position in the glossary. To facilitate browsing documents on the world-wide Web, browsers use a document formatting language called HTML (Hypertext Markup Language) along with an addressing system consisting of the URL (Uniform Resource Locator) to present and navigate hypertext documents. Many browsers based on this standard are available, including Netscape Navigator [Nescape97], Microsoft Internet Explorer [Microsoft97] etc. In addition to supporting the navigation of hypertext links, they support a common model of navigation *history*, which preserves the reader's location as a link is followed. A hypertext document with hyperlinks is shown as figure 10 [JS91].

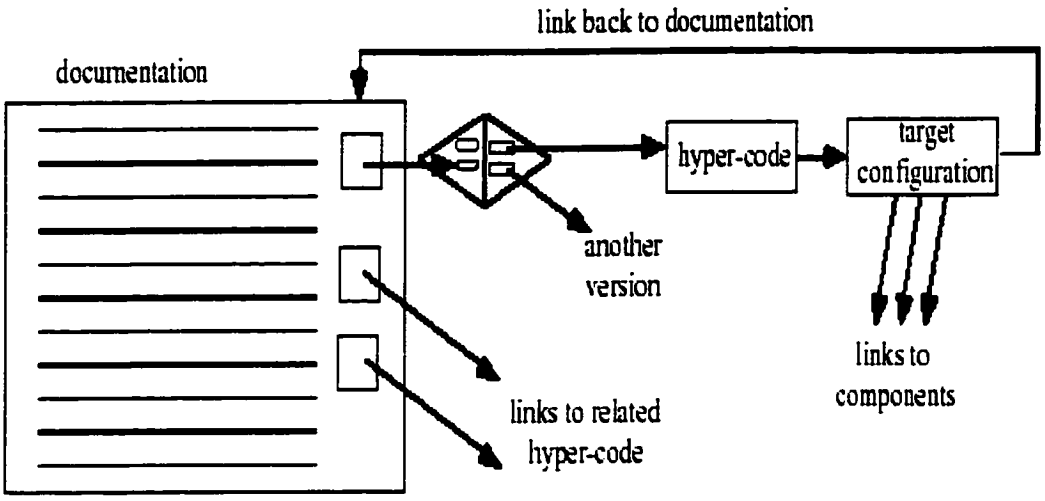


Figure 10. A hypertext Document With Hyperlinks

3.2.2 Focus + Context (Fisheye views)

Fisheye views using focus + context techniques allow people to see both a focus region and the surrounding context in the same window [GUT02]. Interactive focus + context techniques show both local detail and global context in the same view (e.g. [JRP95], [YM94]). They provide a user-controlled focus point for indicating which part of the data is to be shown in detail [GUT02]. Although the computer screen is relatively small, focus + context techniques make it possible to display much information and details which would completely overwhelm the user.

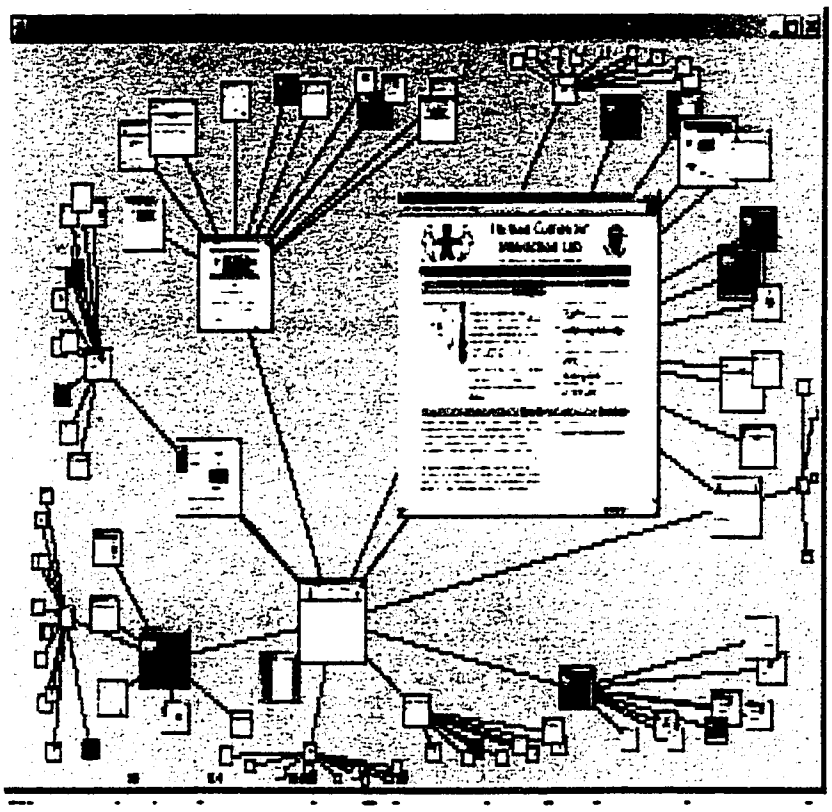


Figure11. An example of fisheye view

These techniques are a solution to the space problem in information visualization, and allow more objects to be displayed than would be possible in an undistorted view [GUT02]. The inspiration for this style of display comes from the view through a fisheye lens. The center of the display is visible in detail, but away from the center the display gradually distorted and compressed. The results of display are similar. A number of designs are applied to focus + context techniques, including the bifocal lens [YM94], the DragMag magnifier [WL95], the rader view [SHH98] etc. An example fisheye system is shown in Figure 11.

3.2.3 Shading/ Color/ Texture

Shading is a powerful technique for creating computer graphics and production animation. Cook's *shade trees* [Coo84] were the base of many later work on shading. He turned simple expressions, describing the shading at a point on the surface, into a parse tree that was interpreted. He introduced the name appearance parameters for the parameters that affect the shading calculations. He also proposed an orthogonal subdivision of types of programmable functions into displacement, surface shading, light, and atmosphere [OL98]. In pattern recognition area, Marc Olano and Anselmo Lastrs produced a shading language and shading language complier for their high end graphics machine PixlFlow[OL98]. PixFlow is hardware, which consists a set of nodes. Each node is essentially a complete computer. The shading language is called *phman*. The advantage of shading language for procedural shading is that the implementation details are hid from *shader*-writer. Shaders that are written and compiled in their shading language, will generate high quality images on the PixFlow.

Color is playing a dynamic role in the application of computer graphics to experimentation and research in the Sciences as well as in the Arts. Major color theorists such as Munsell and Ostwald have developed three-dimensional models that organize "all possible colors" into a system, categorizing colors by three basic color characteristics; hue, value and chroma [Tru81]. Hue refers to red, blue, green or yellow. Ostwald describes hue as distinct color such as redness or blueness [Tru81]. Value refers to the value of color such as lightness or darkness, chroma as the strength or weakness of the color [MUN71]. In information domain, color is a technique used to convey yet more

information. Color can be used to redundantly encode size, with the brighter colors denoting larger components.

Specially, a perceptually uniform color spectrum [LH92] [LHMR92] is used to encode information, and other spectrums with fewer, greater, or differing colors are used to make distinctions between objects. In Tuceryan and Jain's Texture Analysis [TJ98], we can find several definitions of texture. Texture is difficult to define and the difficult is demonstrated by the number of different texture definitions attempted by vision researchers [TJ98].Coggins [Cog82]has compiled a catalogue of texture definitions in the computer vision literature and we give some examples here.

- "We may regard texture as what constitutes a macroscopic region. Its structure is simply attributed to the repetitive patterns in which elements or primitives are arranged according to a placement rule."[TYM78]
- " A region in an image has a constant texture if a set of local statistics or other local properties of the picture function are constant, slowly varying, or approximately periodic."[Skl78]

Texture is formulated by different people depending upon the particular application and there is no generally agreed upon definition. Texture has applications in image processing, document processing and remote sensing. Details are discussed in [TJ98].

Applying in combination with 2D/3D and animation space, shading/color/texture it can be used to display the static view of source and specifications. For example, in focus + context tree, shading can be used to shading the relevant contexts around the focus. On the other hand, shading/color/texture can be used to display the dynamic behavior of program executions, e.g., to represent the couplings of different objects.

3.2.4 Dimension 2D Versus 3D

It is often said that a picture is worth a thousand words [Knight98]. A 2D picture or a 3D picture containing much knowledge will aid visualization of knowledge. For many years the focus of software visualization has been 2D dimension. In the early 1990s, Tyler and Clarke (TC90) realized that a pair of random dot stereograms can be combined together, the results being called "a simple version of Stereogram" (SIRDS), or more generally, an autostereogram. This technique makes use of the brain fusing two images together to present a three-dimensional image in a 2D printed page. It is achieved by using the correlations of pixels in the horizontal direction. Using the correspondences between pixels inhuman brains or computer algorithms, surfaces can be reconstructed from autostereogram it became very popular. The Tree visualization technique with tree-maps is a typical approach to display information in two- dimension (2D). For example, quad-trees [NS84], XY-tree [GJS91] are applications of tree-map visualization; the graphs are displayed in two dimensions. Figure 12. Simulating a solid object on a 2D image plane [TIW94].

The advantage of 3D dimension is that 3D has one more dimension. 3D is more suitable that 2D to visualize abstract data and the amount of information. Further more, the success of the World Wide Web (WWW) has made available a vast amount of abstract data that needs to be visualized. ConeTrees are a typical approach of 3D dimension display. Robertson, el al. describe a 3D visualization techniques for the display of large tree structures [MS02]. This algorithm arranges the tree nodes in 3D space, displaying the structure in perspective. Displaying parent-child relations in the form of a cone makes the graph layout in 3D space. The parent node is placed at the tip of the cone, and the node's

children are distributed evenly around the circular end of the cone. The cones are either arranged vertically (with the root node at the top) or horizontally (with the root node at the left). The user is allowed to interact with the display by smoothly rotating the cones around their axis of symmetry, allowing the cones to be viewed from all sides. From a visual aspect, the 3D dimension display allows the user to pick any node and rotate the cone tree so that the chosen node is brought into the front. An example of cone tree displaying the images in 3D is in figure 13 [Knight98].

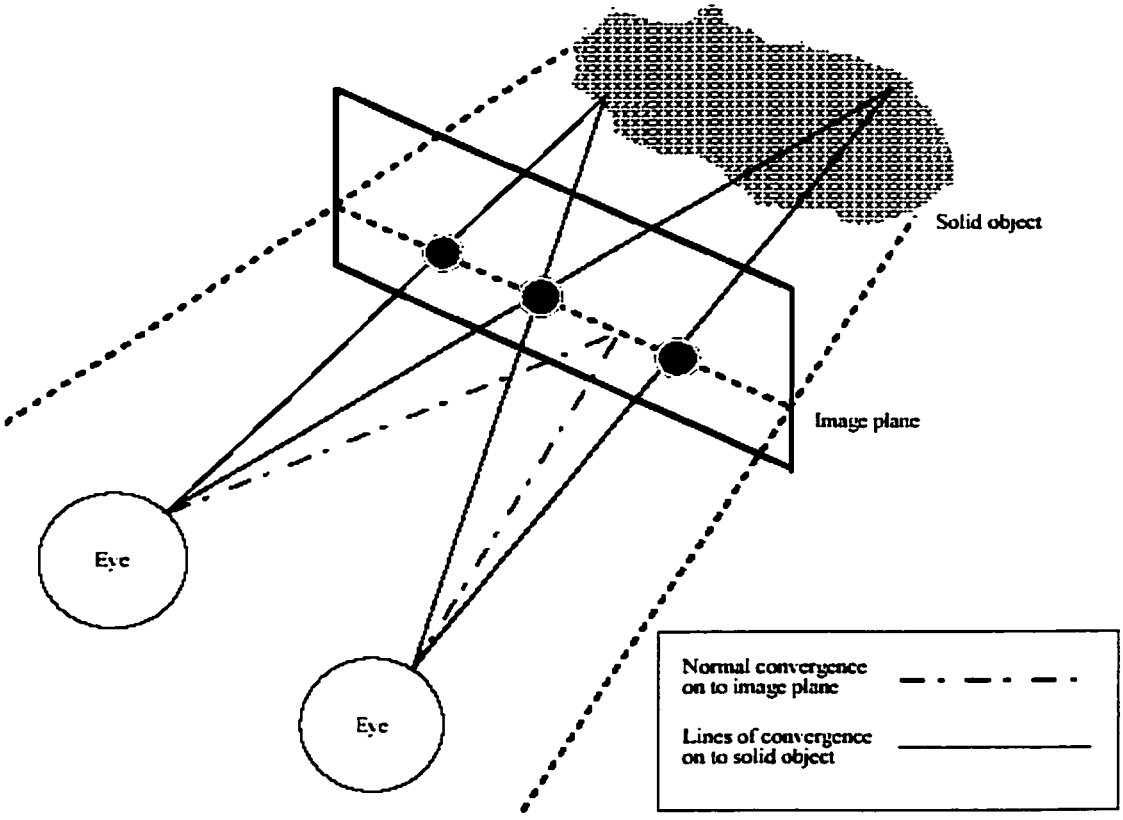


Figure 12. Simulating a Solid Object On a 2D Image Plane

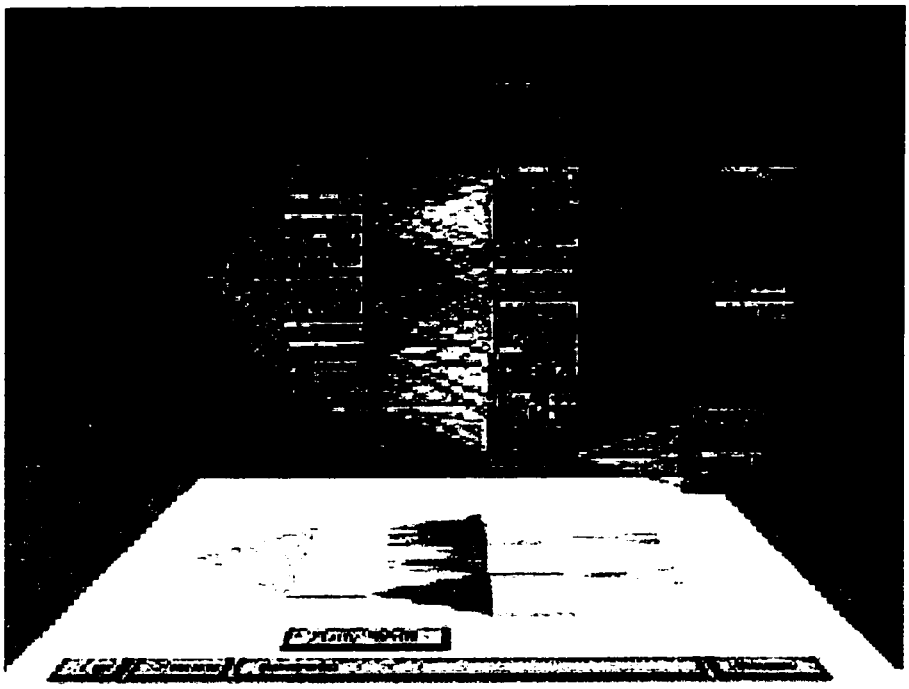


Figure 13. An example of horizontal cone tree from InXight

3.2.5 Animation

In the early part of this century, the art and craft of animation was being refined. Much of the best work was being done at Disney Studios. Frank Thomas and Ollie Johnston, pioneering Disney animator and chroniclers of this area of Disney Animation in *The Illusion of Life*: *Disney Animation*. The animators continued to search for better methods of relating drawings to each other and had found a few ways that seemed to produce a predictable result. They could not expect success every time, but these special techniques of drawing a character in motion did offer some security. As each of these processes

acquired a name, it was analyzed and perfected and talked about, and when the new artists joined the staff they were taught these practices as if they were the rules of the trade. To everyone's surprise, they became the fundamental principles of animation. (Thomas and Johnston, 1984)

Thomas and Johnston (Tomas and Johnston, 1984) enumerated twelve " principles of animation" : squash and stretch, anticipation, staging, straight ahead action and pose to pose, follow through and overlapping action, slow in and slow out, arcs, secondary action, timing , exaggeration, solid drawing, and appeal. John Lasseter outlines the principles of traditional animation, and how they apply to 3D computer animation, in (Lasseter, 1987).

On a computer screen, once a new zoomed geometry is computed, it must be displayed to the user. Changing the display suddenly from old geometry to the new geometry with an immediate transition might place a heavy cognitive load on the user who tries to mentally track the changing positions of the nodes. Animation, In contrast, excels at providing enough information for the audience to follow the action without ever being startled and confused by puzzling behavior. A critical function of the zoom display is that the transitions between zoomed geometries be smoothly animated. Through the animation, the focal node can be seen to grow, while the others shrink to accommodate the focus, visually cuing the new locations of the various nodes. Animation provides the visual cues necessary to understand what is happening before, during, and after the action. By now, animation has been used in the work to illuminate change, for example, in data visualization [MYE90, BOO91, FOR93], algorithm animation and program visualization [LR94, CRM91, LM94].

3.3 Software Visualization

Software visualization can be seen a specialized subset of information visualization, because information visualization is in the process of creating a graphic representation of abstract, non-numerical, data [KNI98]. In the area of computer science, a better definition is provided by Claire Knight. She defines:

" Software visualization is a disciple that makes use of various forms of imagery to provide insight and understanding and to reduce complexity of the exiting software system under consideration." [KNI98]

From that definition, we can describe the goal of software visualization is for understanding of software system and reduce the complexity of software, because it has been known the software itself is real complex. We can have many views from a program. Myers calcifies them in [MYE90]:

- *Static code visualization*
- *Dynamic code visualization*
- *Static data visualization*
- *Dynamic data visualization*
- *Dynamic algorithm visualization*

In this section, I apply a static and a dynamic view to category software visualization technique. More specially, I distinguish between dynamic view from a forward engineering perspective and dynamic view from a reverse engineering perspective.

3.3.1 Static View

For fully understanding software both static and dynamic views are necessary. A Static view of a program is based on the static structure of the source code. There are often said to be three primary aspects of a system apart from its identity. These are respectively concerned with: a) data, objects and their structure; b) architecture or a temporal process; and c) dynamics or system behavior. A static view refers to a) and b), there are software artifacts and their relations. The particular visual constructs used to represent, for example, classes, member functions, loops, and branching, are primary based on empirical studies of programmers [FOR93] and widely accepted sources such as Booth [BOO91, 93]. In Java, for example, such artifacts could be classes, interfaces, methods, variables etc. The relations might include extending relationships between classes or interfaces, method call between methods, containment relationships between classes and methods or variables etc.

3.3.1.1 Structural

Software architecture [GS93] is the organization of a software system as a collection of components, connections between the components, and the constraints on how the components interact. In object-oriented methods aspects could include the object structures, the structure of classes and their relationships, the structure of inheritance. Object structures means how objects relate to each other. A class is a collection of a set of objects and contains attributes and operations. The relationships describe the ways a class interacts with other classes. Inheritance is one of the characteristics of object-oriented methods. In object-oriented programming, a class can create instances of itself in

memory. These instances 'inherit' exactly all the features of the class: its methods and attributes. An instance can be a member of only one class. It is possible for classes to inherit all the features of more general classes. For example, classes **Cars** and **Cycles** specialization of the class of **Vehicles**. Inheritance can be single or multiple. Type inheritance can be different from class inheritance, because in type inheritance only the specification and no implementation are inherited.

3.3.1.2 Use Case Diagram

A use case is a goal-oriented collaboration between a system and an actor; where an actor is a user adopting a role. A use case diagram shows a collection of use case and external actors that interact with the system. A use case describes the interactions and behavior of a system during an entire transaction that involves several objects and actors. Within a use case model, relationships between use cases can be the model, i.e. a use case can include other use cases as part of its behavior description. The specification of the external behavior of a use case may be given by a state diagram. The implementation of a use case diagram can be described by a collaboration diagram. Using use case diagrams, the relationships between actors and use case diagrams within a system are tracked. Figure 14 show a use case diagram for renting and paying bill in a video renting system.

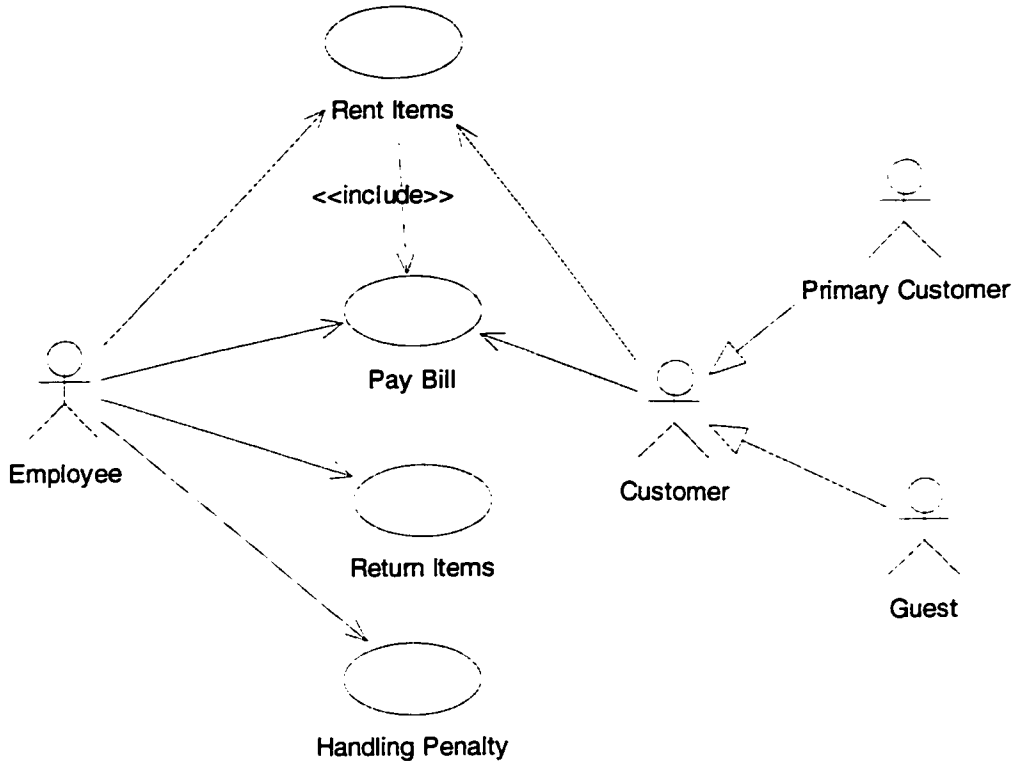


Figure 14. A use case diagram for renting and paying bill in a video renting System

3.3.1.3 Class Diagram

A class diagram 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. The subsystem diagram describes the classes included in each subsystem and the relationships between the subsystems, as well as the relationships

between the classes within the same subsystem. The full class diagram gives an overall view of the classes and relationships of the System, and the class description forms give a detailed description about these classes. Class diagram shows classes and the relationships between classes. Figure 15 show a class diagram for class customer in a video renting system.

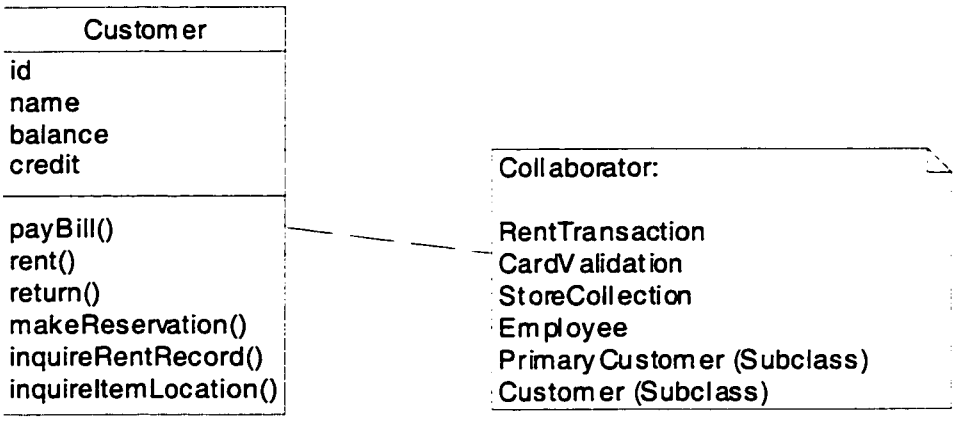


Figure 15. A class diagram for class customer in a video renting system

3.3.1.4 DFD (Data Flow Diagrams)

Data Flow Diagram (DFD) is a means of representing a system at any level of detail with a graphic network of symbols showing data flows, data stores, data processes, and data sources/destinations. The purpose of data flow diagrams is to provide a semantic bridge between users and systems developers. The diagrams are:

- Graphical, eliminating many words;
- Logical representations, modeling WHAT a system does, rather than physical models showing HOW it does;

- Hierarchical, showing systems at any level of detail; and
- Allowing user understanding and reviewing.

The goal of data flow diagramming is to have a commonly understood model of a system. The diagrams are the basis of structured systems analysis. Data flow diagrams are supported by other techniques of structured systems analysis such as data structure diagrams, data dictionaries, and procedure-representing techniques such as decision tables, decision trees, and structured English.

3.3.1.5 Treemap

Tree visualization technique with tree-maps is a typical traditional approach to display hierarchy information. Johnson, el al. describe the visualization techniques which supports the display of hierarchical trees, and maximizes the uses of the available screen space [Shn91]. The original motivation for the technique was to display the complex hierarchical file structure of a hard disk drive. Ben Shneiderman [Sam89] describes tree structure as a rooted, directed graph with the root node at the top of the page and children nodes below the parent node with lines connecting them, tree structure as shown in Figure 16.

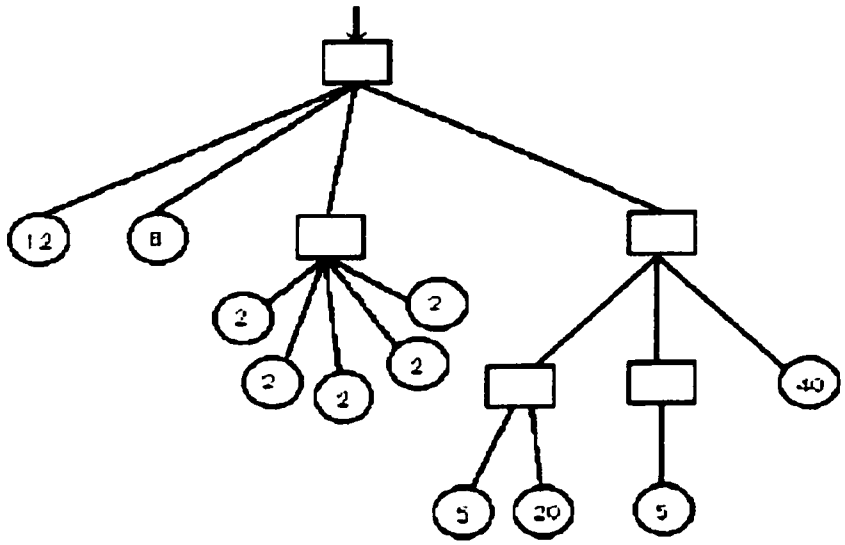


Figure 16. Typical 3- level tree structure with numbers indicating size of each leaf node

Research on relationships between 2D images and their representation in tree structures has a focus on node and link representations of 2D images. This work includes quad-trees [NS84] and their variants that are important in image processing. The goal of quad trees is to provide a tree representation for storage compression and efficient operations on bit-mapped images. XY-tree [GJS91] is a traditional tree representation of two-dimensional layouts found in newspaper, magazine, or book pages. Typically, tree-maps are a representation designed for human visualization of complex traditional tree structures: arbitrary trees are shown with a 2-d space-filling representation [Sam89]. Figure 17 is the tree-map of figure 16 [Sam89].

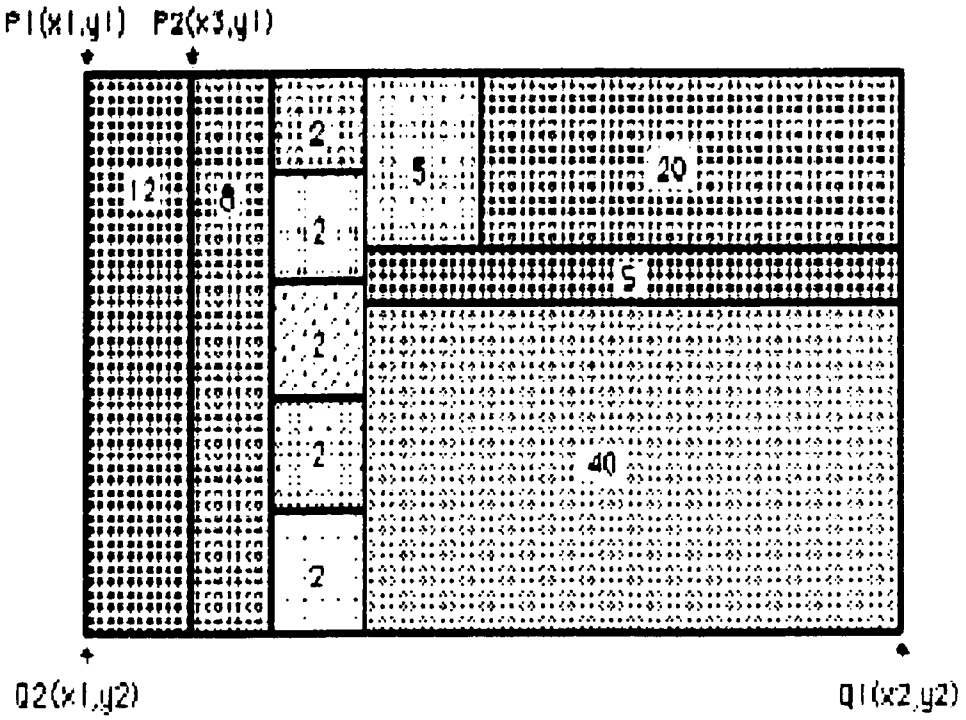


Figure 17. Tree-map of Figure 16.

3.1.1.6 Hyperbolic Space

Lamping and Rao [LR94] describe a technique for displaying and manipulating large hierarchies through the use of hyperbolic geometry. The hierarchy is represented on a hyperbolic plane, and mapped to a circle on the Euclidean plane. The characteristics of the resulting display are reminiscent of the view of a true optical fisheye lens: the center of the circle is in detail, and the edges show minimal detail. The display also supports a single focus, although a great deal of the hierarchy near the focus is also visible. The technique also suffers from the corner screen space neglect that is characteristic of all circular displays.

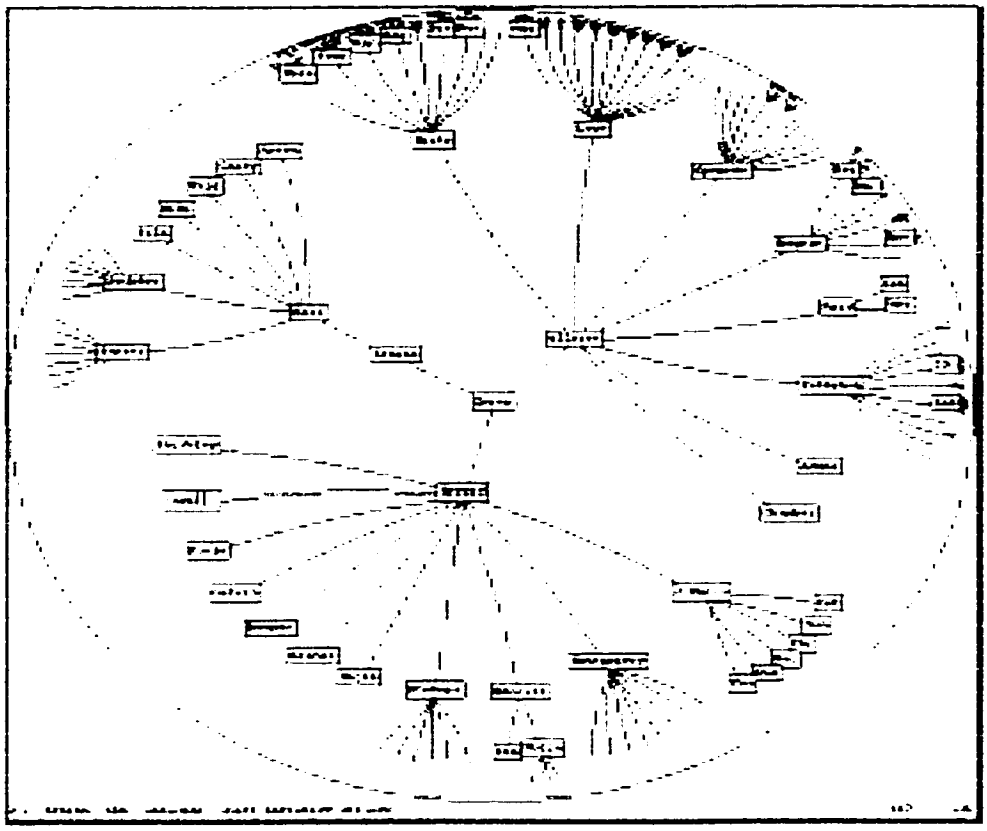


Figure18. An Example of Hyperbolic Browser

Figure 16 [LR94] is an example of hyperbolic browser. The hyperbolic browser supports interaction with larger hierarchies than conventional browsers with modest computational requirement, it is more effective navigation around the hierarchy [LR93].

3.3.2 Dynamic View

Dynamic view refers to the run-time behavior of program. Programs can have wildly different behavior over their run time, and these behaviors can be seen even on the largest of scales (over the complete execution of the program). Dynamic view contains software

artifacts as well. In addition, it contains sequential information, information about concurrency and code coverage, etc.

3.3.2.1 Behavior – show dynamic system behavior from a forward engineering perspective

Forward engineering, the obvious opposite of reverse engineering, is referred to, to distinguish the traditional software engineering process from reverse engineering [CC90]. The direction of a forward engineering is from design to implementation. First, system behavior is illustrated in a problem domain and the solutions are designed. Finally, the solutions are implemented into a system by source code. In order to further understand the behavior of a system, we focus on an object-oriented (OO) system. Catalysis recommends the micro-process for system and component specification and design illustrated in figure 19 [Grah01]. The design can begin either with actions or with objects. Post-conditions are written on the actions, and then the vocabulary that the type model must clarify is teased out. Now the techniques of snapshot, sequence and state diagrams are used to clarify and refine the models, leading to new and additional actions. Each iteration, introducing more detail and eventually moving from a specification to a design and implementation.

The important design principles that should be applied during the process are follows [Grah01]:

- Assign responsibilities to objects evenly, trying to get objects of roughly similar size.
- Ensure that all associations are directional.

- Clarify invariants by using snapshots.
- Avoid circularity and high fanouts.
- Make sure that dependencies are layered.
- Minimize coupling and visibility between objects.
- Apply patterns throughout the design process.
- Iterate until model is stable or deadline approaches.

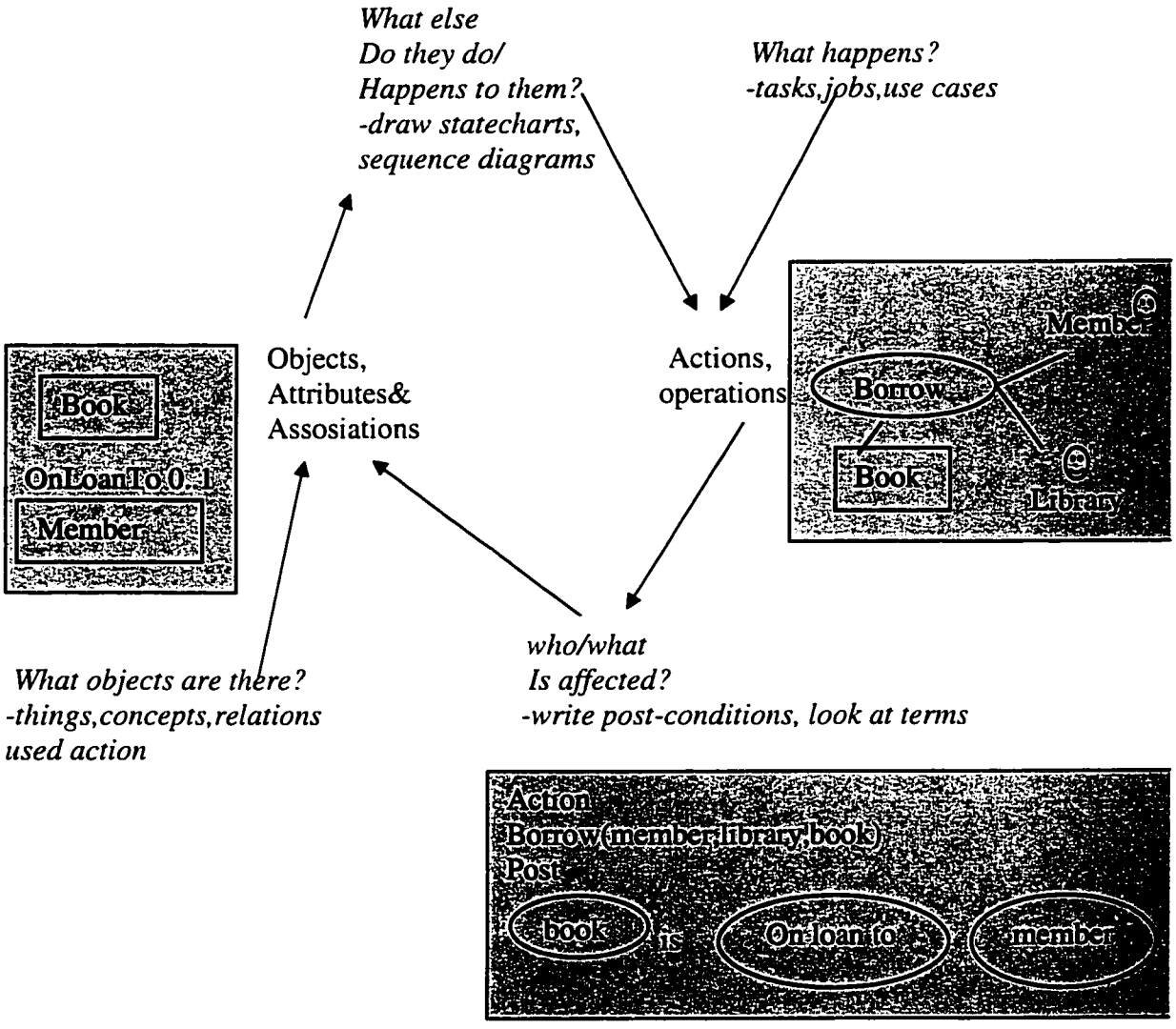


Figure 19. The Catalysis specification and design micro-process [Grah01]

3.3.2.1.1 Sequence Diagram

An *interaction diagram* shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them [Booch90]. Interaction diagrams include sequence diagrams and collaboration diagrams. A sequence diagram emphasizes the time ordering of messages [Booch90]. In addition, a sequence diagram may show the lifelines of the objects involved in the interactions. An *interaction* is a behavior that comprises a set of messages exchanged among a set of objects within a context to accomplish a purpose [Booch90]. A message is a specification of a communication between objects that conveys information with the expectation that activity will ensure [Booch90]. Graphically, a sequence diagram is a table that shows objects arranged along the X-axis and messages, ordered in increasing time along the Y-axis.

A sequence diagram is developed on the basis of the use cases. Figure 20 shows a sequence diagram for rent items in a Video Renting system. In a sequence diagram, a message is being sent between objects.

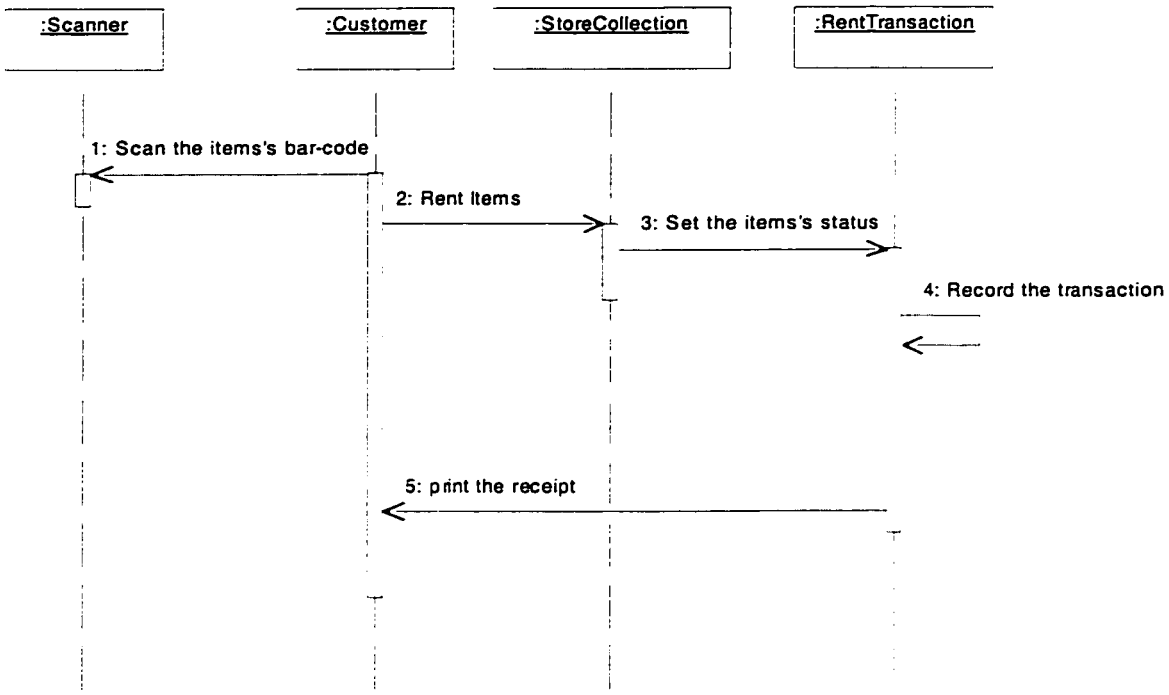


Figure 20. A sequence diagram for renting items in a video renting system

3.3.2.1.2 Collaboration Diagram

Collaboration diagram are a type of interaction diagrams that emphasizes the structural organization of the objects that send and receive messages [Booch90]. Collaboration diagrams depict objects and links between the collaboration of objects. Links visualize the message flow between the corresponding objects. Messages may have an argument list and a return value. Message ordering in the overall transaction is described by a modified Dewey decimal numbering, specifying the sequential position of a message within its corresponding thread. A composite object is an instance of a composite class that implies an aggregation between the class and its part. Parameterized collaborations

represent design patterns that can be used repeatedly in different designs. Figure 21 is a collaboration diagram of renting item in a video renting system.

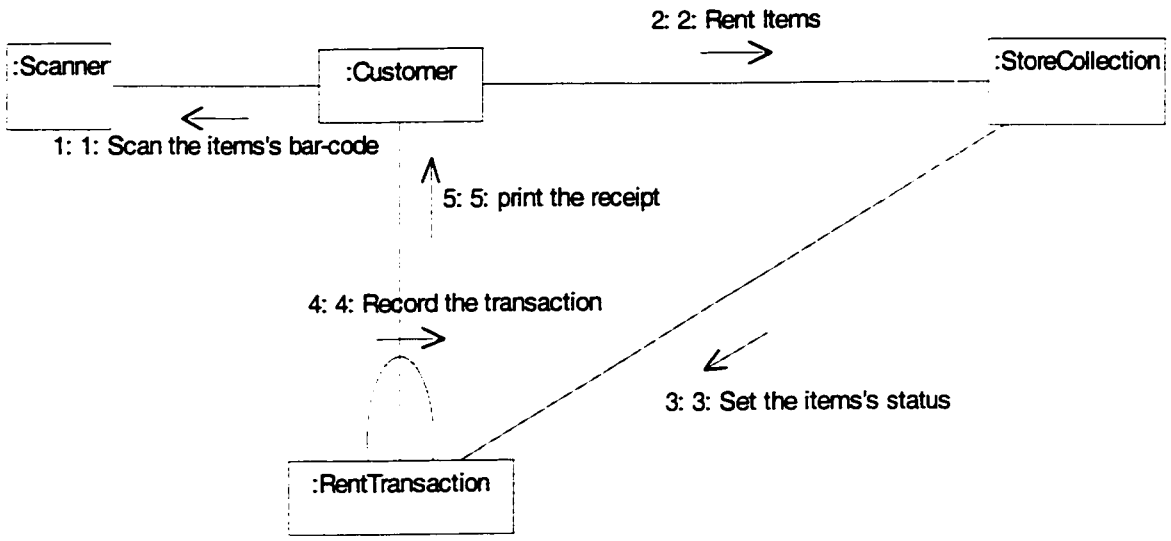


Figure 21. A collaboration diagram of renting item in a video renting system

3.3.2.1.3 State Diagram

State diagrams are based on the statecharts derived by Harel [Harel87]. They are similar to the state-machine diagrams used in OOA/OOD and OMT. They describe the reaction of an object, in reply to events received, in the form of responses and actions. State diagrams basically consist of states and state transitions. A state represents a condition during the existence of an object in which it waits for an event to be received, performs some action or satisfies some condition. An event is an occurrence that may trigger a state transition.

- State Transition Diagrams

State transition diagrams have been used right from the beginning in object-oriented modeling. The basic idea is to define a machine that has a number of states (hence the term finite state machine). The machine receives events from the outside world, and each event can cause the machine to make a transition from one state to another. Thus you can get a good sense of what events should occur, and what effect they can have on the object. State Diagrams can be created for each of the classes and the events that will change the states will be described and the appropriate operations that should be done when an event is triggered. Also guard conditions for each event and the description of internal operations for each state will be described.

- Activity Diagrams

Activity diagrams are a special case of state diagram that are to be used in situations where most of the events represent the completion of internal-generated actions. Activity diagrams are one of several ways to model the dynamics of a system. An Activity diagram is basically a flow chart that describes the flow of control from one activity to the next. One can show sequential and/or concurrent steps of a process, model business workflows, model the flow control of an operation, or the flow of an object as it passes though different states at different points in a process. Unlike interaction diagrams (Sequence, Collaboration) that emphasize the flow of control between objects, Activity diagrams emphasize the flow of control between activities. An activity can be described as "an ongoing, non-atomic execution within a state machine" and the ultimate result is some action that affects the state of the system or returns some value. The Scope of the activity diagram depends on what we decide. We can decide to use a single use case, a

portion of a use case, a business process that includes several use-cases or a single method of a class. Figure 22 shows an activity diagram for use case: Rent Item

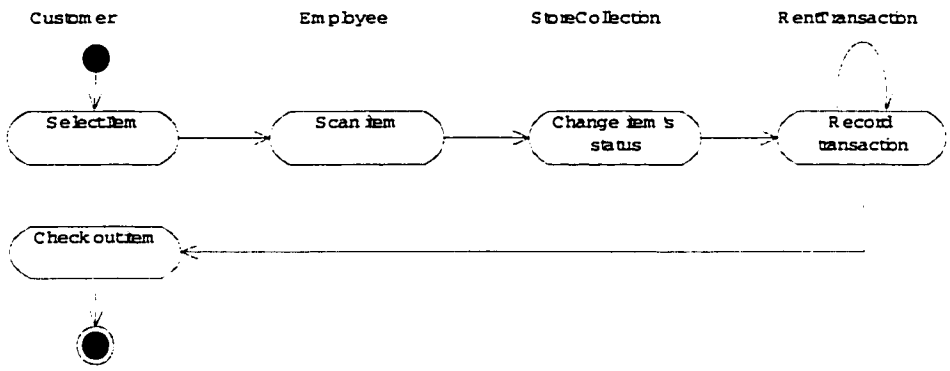


Figure 22. An activity diagram for use case: Rent Item in a video renting system

3.3.2.2 Program Executions - program executions from a reverse engineering viewpoint

Reverse engineering has many tasks. [TilleyTilley98] discusses several of the most important: program analysis, plan recognition, concept assignment, redocumentation, and architecture recovery. The first three tasks can be viewed as pattern matching at different levels of abstraction.

- Program analysis: is for source code analysis and simple code restructuring purposes. Abstractions could be done in different levels. For example, Ning [Ning89] identified four levels of abstraction for reverse engineering: implementation, structural, functional, and domain.

- Plan recognition: attempts to discover instances of abstract representations of commonly used algorithms and/or data structures in the subject system.
- Concept assignment: is the task of discovering individual human-oriented concepts and assigning them to their implementation-oriented counterparts in the subject system.

Reverse engineering aides program understanding starting from the source code that implements a system. Program executions are the dynamic behaviors of a system. The direction of reverse engineering for program understanding is from implementation to a design-level behavioral model. The gap in terms of abstraction between design-level behavioral models and the source code that implements a system happens often. The gaps can result in improper mapping from design to implementation. Sequence diagram and collaboration diagram can record the dynamic behavior of a system during program execution. By comparing the abstract behavior of a real system with the design level information, these diagrams provide for a detailed analysis of the dynamic program behavior.

3.3.2.2.1 Sequence Diagram

In a sequence diagram, objects are arranged along the X-axis, and messages are ordered in increasing time along the Y-axis. During the program execution, sequences of interaction between classes and objects reoccur. By comparing observed behavior and designed behavior, a sequence diagram helps keep documents updated. For example, in the Video Renting System we discussed before, the Sequence diagram for RentItem have objects: RentTransaction, Customer, StoreCollection. During the program execution of RentItem, related methods are:

- Related methods of Customer: Rent( );
- Related methods of StoreCollection: Set( );
- Related methods of RentTransaction: Set( );

A Message is passed from one object to another, and related methods are called up. The uses of sequence diagrams for software visualization are:

- Construct design-level behavioral models from low-level behavior of a system during the program execution.
- Compare observed behavior and predicted behavior. The results will help validate the design/implementation and help reengineering.
- Keep the behavioral model up to date with the respect to modification of a system's implementation.

3.3.2.2.2 Collaboration Diagram

Collaboration diagrams emphasize the structural organization of the objects that send and receive messages. The order and frequency of execution of the various subroutines can be observed. This information will help decide the order in which to study the piece of code. In the design level, we just predict the behaviors of the system. In the implementation level, we can validate the behaviors of the system. A collaboration diagram is like a call graph, it is a compact representation of calling behavior that summarizes all possible run-time activation stacks. Collaboration diagrams are dynamic behaviors containing direct or indirect program executions. For example, in a *for* loop, we can observe the run-time frequency of the loop execution and the other calls during

the loop execution. The analysis of results will help indentify design issues that may be difficult to be observed.

4. Survey of Software Visualization Tools

The major goal of visualization tools is to extract abstractions from software representations and transfer this information into the minds of software engineers for software evolution purposes [MWT94]. Tilley suggests information exploration " *holds the key to program understanding*". Some visualization tools show the animations of algorithms and data structures; some show dynamic program executions for understanding run-time behavior; some tools show textual representations of source code and documentation. Of key importance is whether such a tool supports bottom-up comprehension, top-down comprehension or some combination of two [SFM99]. Also important, especially for large systems, is how the maintainer browses or navigates the visualization [SFM99].

4.1 Visualization Tools For Bottom-Up Program Comprehension

Bottom-up strategies can be supported by providing direct access to the lowest level of program details. The details are normally found at the statement level of the program source code, which should be made accessible to the user. Through such tools, the users should navigate the source code of larger programs easily. According to Storey et al. [SFM99], some features of visualization tools that enhance bottom-up comprehension are:

- To indicate syntactic and semantic relations between software objects
- To reduce the effect of demoralized plans
- To provide abstraction mechanisms

4.1.1 VIFOR and VIFOR 2

VIFOR stands for **V**isual **I**nteractive **FOR**tran and is a software tool that is geared towards maintaining Fortran 77 code [RDL90]. VIFOR offers multiple views of source code either in a textual form of the code or a graph layout. The Layout mechanism of graphs such as call graph and data dependency graphs attempts to be standard. VIFOR is one representative of the software visualization tools developed in last ten years. Early work of C program maintenance and comprehension tools are based on VIFOR. VIFOR 2 [RA96] is an extension of VIFOR, which is written in languages C and Fortran, and HMS. VIFOR 2 combines two technologies: browsing and hypertext documentation. VIFOR2 supports browsing PAS documentation, and help recording the understanding of the system continually. The partitioned annotations of software (PAS) proved to be a superior way to document software [RGD94].

4.1.2 POLKA

The POLKA (Parallel program-focused Object-oriented Low Key Animation) [SK93] was created at Georia Tech, under the guidance of Dr.John Stasko. The system not only allows students to watch algorithm animations that were created previously but also lets them build their own animations. POLKA is a general-purpose algorithm animation system that provides support for color, real-time, 2D/3D dimension smooth animations. 2D is implemented on top of the X Window System and 3D is implemented on top of Silicon Graphics GL in C++. The POLKA is an object-oriented basis of visualization and animation that includes high-level graphical object and motion primitives. Two features of POLKA [SK93] are:

- *True* animation – POLKA offers smooth, continuous movements and actions, not just blinking objects or color changing.
- *Concurrent* – POLKA offers overlapping animation actions that can properly reflect the concurrent operations occurring in a parallel program.

The 3D version provides many default parameters and simplifications so that the designers need not worry about the details of graphics. Programmers need not understand 3D graphics techniques like shading, ray-tracing etc., in order to create a 3D visualization. POLKA provides its own high-level abstractions to make animations easier and faster to create.

4.1.3 ANIMAL

ANIMAL [RSF00] is a flexible and powerful algorithm animation tool, developed at the University of Siegen worked. The main purposes of ANIMAL are to display the animations of algorithms, data structures rather than using sliders and blackboards for lectures. ANIMAL is written completely in Java using Java's Swing library. ANIMAL supports three separated approaches for generating animations: *visually*, *by scripting*, and *by API call* [RF00]. The interface of ANIMAL is user friendly. Animation authors do not need programming knowledge and can generate and edit the animations in a drawing pane *visually*. ANIMALSCRIPT is a simple, line-based language driven by text commands [RF00]. That means:

- Each command must be given on a *single line*. This also means that the lines may become somewhat long, so that users should refrain from using auto wrapping in their editors.

- All entries are provided as ASCII text,
- And each line starts with a unique command, making it easy for ANIMALSCRIPT to parse the line.

The API generates animations files in ANIMALSCRIPT language. ANIMAL supports the integration of both source/pseudo code and text descriptions into animations. Both ANIMALSCRIPT commands and ANIMALGENERATOR API method calls are easily accessed by the authors, and providing dynamic generation of program visualization.

4.1.4 SeeSys

SeeSys [BE94] is a visualization system that uses a space-filling technique to display complex software system effectively. The technique of SeeSys is to visualize statistics of program source in subsystems, directories, and files. User interaction, screen real estate, spatial relationships and color are incorporated into systems to layer additional information onto the base display. Interactively users can choose a particular subsystem they want to look at by using their mouse. In the computer screen, rectangles are placed next to each other so that 100% of the display area is utilized. Zoom-in is possible when the subsystem is too small. Spatial relationships mean users with a slide bar can optimally determine the number of rows used, and algorithm used to equalize heights of rectangles. Finally, color is used to redundantly encode size. Three principles should be adhered when creating visualizations of larger software systems [BE94]:

- The individual components can be assembled to form the whole. This allows the user to easily see the relationships between them.

- Pairs of components can be compared to understand how they differ.
- The components can be disassembled into smaller components. This important feature of the components allows the structure of the display to reflect the structure of the software.

SeeSys scales up to complex software system and allows users to view the whole system, providing information about the real problem of the system.

4.1.5 Rational Rose

The Unified Modeling Language [BRJ99] is a set of description techniques suited for specifying visualizing and documenting an object-oriented system. It is UML's intention to produce a single, common, and widely usable modeling language for these methods and, working with other methodologies, for other methods as well. The UML is an evolution of general-purpose, tool-supported, standardized modeling language which has features of Metamodel, graphic notation for visual representation, Extension Mechanisms and so on. The UML specification consists of two interrelated parts that are UML Semantics and UML Notation. UML Semantics is a mental model that specifies the abstract syntax and semantics of UML object modeling concepts. UML Notation is a graphic notation for visual representation of the UML semantics. The architecture of the UML is based on a four-layer metamodel structure, which consists of the following layers: user model, model, metamodel, and meta-metamodel. Like other software engineering 'methods' UML provides a set of graphical and textual modeling techniques that aim to be understood by system developers and customers. Each technique is used to model the system from a number of different perspectives. For example, class diagrams

are used to model static(data) properties; sequence diagrams are used to model the data flow of messages between objects.

UML is a tool-supported language that has the ability to visualize and express the system's structure and behavior at many different levels of abstraction. For example, Ration Rose [Ratinal00] supports reverse engineering of, e.g., C++ and Java software systems. When reverse engineering a Java program, Rose constructs a tree view that contains classes, interfaces, and associations found at the highest level. Methods, variables etc. are nested under the owner classes. Rose also constructs (on demand) a class diagram representation of the extracted information and generates a default layout for it. Additionally, Rose automatically constructs a package hierarchy as a tree view. Rose is able to reverse engineer the information from source code (.java files), byte code (.class files), jar files, or packed zip files. Java reverse engineering module can be given instructions on files, directories, packages, and libraries to be examined.

4.1.6 Features of Visualization Tools Enhancing Bottom-Up Comprehension

Bottom-up comprehension is based on chunking. The bottom-up strategy of program understanding is to start from the smallest chunks until chunks of the program big enough emerge. In this section, we summarize some features of visualization tools that enhance bottom-up comprehension. A comparison table is shown below.

	Multiple Views	Decomposition	Provide Abstraction Mechanisms	Navigation
VIFOR and VIFOR2	Yes Text form, graph layout	No	Yes Partitioned Annotation of Software (PAS)	Yes Hypertext links , browser
POLKA	Yes View of data structure, view of algorithm animation	No	yes Properly reflecting the concurrent operations that occurring in a parallel program	No
ANIMAL	Yes Animation of algorithms, data structures	No	Yes Three approaches to generate animation	No
SeeSys	No	Yes By using moose	Yes By comparing pairs of component	Yes Zoom in to see a small subsystem, a slide bar determine spatial relationships
Rational Rose	Yes UML diagrams	Yes UML diagrams	Yes UML diagrams	Yes UML diagrams

Table 4.1 Features of visualization tools that enhance bottom-up comprehension

(Continue)

	Usability	Source Code Analysis	License
VIFOR and VIFOR2	Yes Hypertext links , browser	Hypertext documentation	Shareware
POLKA	Yes 3D version providing default parameters and simplification	Algorithm animation	Open source
ANIMAL	Yes ANIMALSCRIPT	Algorithm animation	Freeware
SeeSys	No	Decomposition	Commercial
Rational Rose	No User has to learn before using	Yes UML diagrams	Commercial

Table 4.1 Features of visualization tools that enhance bottom-up comprehension

Multiple Views: Multiple Views, source code view and graphic view

Decomposition: Decomposition, a complex system can be decomposed into subsystem

Provide Abstraction Mechanisms: Provide abstraction mechanisms, individual components can be chunked to form the whole system

Navigation: Navigation, reducing the effects of decomposition

Usability: Usability, providing both novice and invoice and experienced programmer easier access

Source Code Analysis: Source code analysis, using techniques such as parsing, slicing, data-flow analysis or querying etc. to analyze the source code

License: License, such as freeware, shareware, open source, and commercial

VIFOR is one typical representative of the software visualization tools developed in the last ten years. VIFOR provides multiple views of the source code; Partitioned Annotation of Software (PAS) is used as abstraction mechanism. VIFOR2 is an extension of VIFOR. Two new technologies of browsing and hypertext document are combined in VIFOR2, which makes it user friendlier to use. POLKA and ANIMAL are two powerful algorithm animation tools. Animation tools are used to visualize source code, and the resulting visualization is straightforward. The differences between POLKA and ANIMAL are that POLKA is parallel program-focused reflecting the concurrent operations and ANIMAL uses three approaches to generate animation. POLKA is open source and ANIMAL is freeware. SeeSys is a visualization tool that uses a space-filing technique to display complex system, and decomposition of source code is provided by visualizing code in subsystems, directories, and files. VIFOR / VIFOR2, POLKA and ANIMAL do not provide support for the decomposition of source code. SeeSys uses zooming and a slide bar to navigate the source code to reduce the effect of decomposition. Rational Rose supports visualization of e.g., C++ and Java software systems by using only UML diagrams to visualize the source code structure. Rational Rose provides support for multiple views, decomposition, abstraction mechanism, navigation and source code analysis. Based on its wide spread use in forward engineering, Rational became also a major player in bottom up visualization of source code. It should be mentioned that all of these tools suffer from limitations in reverse engineering certain language constructs. For example they have problems in identifying aggregation constructs. All of the current tools have also limitations with respect to the scalability and their layout algorithms to present large

systems/amount of data. Additionally, these tools do not provide the user not necessarily with adequate levels of abstractions and are limited to 2D visualization.

4.2 Visualization Tools for Top-Down Program Comprehension

In Top-down program comprehension, a hypothesis is created by program experts and then refined until the comprehension process is completed. Visualization tools that enhance top-down program comprehension will support hypothesis construction and verification, then provide a method documenting and linking hypothesis to the relevant parts of source code or document. Some features of visualization tools that enhance top-down comprehension are [SFM99]:

- Support goal-directed, hypothesis-driven comprehension
- Provide an adequate overview of the system architecture at various levels of abstraction

4. 2.1 Hy+

The Hy+ system is a generic visualization tool for visualizing objects and relationships among them [MS95]. Hy+ supports a novel visual language called *Grapglog* [MS95]. *Grapglog* is used for querying database. This is a traditional way of using database queries: the newly defined relationship either gives a direct answer to a user question, or it provides a new view on the existing data. Applications of the tool can be found in identifying *design pattern* [GHJV95] in the code. Design patterns are high-level design descriptions in object-oriented code. Hy+ has the feature of hypothesis-driven comprehension. Hy+ provides a user interface with extensive support for visualizing

structural (or relational) data as *hygraphs* (Consens et al., 1994), an extension of graphs inspired by Harel's higraphs (Harel, 1988). It provides a number of browsers and graphical editors to display and create the graphical presentations and extensively uses color and icon symbols to improve the generated visualizations [HMV95]. Additionally, in Hy+, filtering mechanism can be used to decide what to *show*. Hy+ makes use of fisheye display techniques [MS95]. The derived data is presented in a graph where nodes represent objects and arcs represent the relations of objects.

Hy+ can be in interfaced with a modified web browser Mosaic 2.0 for web visualization [HMV95]. *Web documents* are written in HTML (HyperText Markup Language) and provided by Web servers. The modified Mosaic 2.0 with Hy+ visualization will give users graphical overviews of his or her browsing.

4.2.2 Jambalaya

Jambalaya [SMSBEFN01] is a knowledge acquisition environment. Jambalaya integrates an interactive user interface SHriMP [SWFM97] plug-in with protégé. The protégé environment has been developed as Stanford University over the past 16 years [Protégé, MFGNCG00, NSDCFM01]. Protégé is an ontology editor supporting knowledge acquisition for computer experts in many knowledge domains. Ontology has become an increasing important research topic and has shown its usefulness in application areas such as intelligent information integration, information brokering, or knowledge-based systems, to name but a few [SM00]. Ontology is as a set of definitions of formal vocabulary. Ontology is a description (like a formal specification of a program) of the concepts and relationships that can exist for domain experts. The purposes of ontology are to enable knowledge sharing and reuse, and provide an agreement of a domain

understanding. SHriMP (Simple Hierarchical Multi-Perspective) is visualization for seamless exploring software structure and browsing source code, with a focus on effectively assisting hybrid program comprehension strategies [SWFM97]. A nest graph is used to visualize the structure of software, both pan + zoom and fisheye view approaches are for exploring a nested graph. Program code can be further explored by following hypertext links in the codes or by the pan + zoom technique in order to create multiple views at different levels of abstraction and perspective.

Jambalaya integrated SHriMP plug-in with Protégé, further technical details are available on the website at [ShriMP]. A knowledge base consists of an ontology and a set of instances. A nested directed graph is used to display the knowledge base visually. Nodes represent the instances and corresponding class. Different colors are used to distinguish instance nodes and class nodes. Arcs represent slot dependencies between classes and instances in the knowledge base. Using Jambalaya during knowledge acquisition, the structure of large knowledge bases are understood or maintained.

4.2.3 Rational Rose

Rational Rose [Ratinal00] is a graphical software-modeling tool. Rational Rose uses the Unified Modeling Language (UML) as its primary notation. The Unified Modeling Language (UML) is a general-purpose visual modeling language used specifies, visualize, construct and document the artifacts of a software system. The UML unifies object-oriented analysis and design techniques and their associated diagrams into a common model. The language is based on a small number of core concepts that most object-oriented developer can easily learn and apply. The core concepts can be combined and extended so that expert object modelers can define large and complex system across

a wide range of domain. The UML specification consists of two interrelated parts that are UML semantics and UML notation. UML semantics that is a mental model that specifies the abstract syntax and semantics of UML object modeling concepts. UML notation which is a graphic notation for visual representation of the UML semantics, include class diagram, sequence diagram, collaboration diagram, statechart diagram etc., UML software visualization deriving forward engineering visualization software may include all diagrams and views. Diagrams and views of UML provide two key benefits to the developer. The first is the ability to visualize and express the system structure and behavior at many different levels of abstraction. UML allows user to construct more abstract concepts from more primitive ones and deal with the system at any level of abstraction that we find convenient. The other key benefit is the ability to look at different aspects of the system. We can, for example, look at the requirements, the structural aspects, or the behavior aspects. Each UML diagram has its own advantages that allow focusing on some particular aspect. For example, to abstract structural aspects, we focus on how objects are distributed through the system and the policies, procedures, and structures used to allow them to collaborate. Or, we can focus instead on the large-scale organization of the subsystems and components. Or we can look at how concurrency is managed in the system and analyze it for schedule ability. This is also scarcely possible for source-code based systems.

4.2.4 Features of Visualization Tools Enhancing Top-Down Comprehension

The strategy of top-down comprehension argues that the program experts develop a hypothesis, and then the experts use their experience to confirm or reject the hypothesis. In this paper, we summarize some features of visualization tools that enhance top-down software comprehension. A comparison table is shown below.

	Support Hypothesis-Driven Comprehension	**Different Levels Of Abstraction**	**Relational Views**	**Usability**
Hy+	Yes *Graphlog* supports database querying	Yes Filtering mechanism decides what to show, and *hygraph*	Yes *Hygraph* textual browsing of source code	No
Jambalaya	Yes Protégé is an ontology editor supporting knowledge acquisition	Yes Nested interchangeable views	Yes SHriMP Class hierarchy view is shown on the pane of Protégé editor	Yes Protégé editor is user friendly
***Rational Rose**	Yes UML semantics	Yes UML graphical notations	Yes UML graphical notations	No User has to learn before using it

Table 4.2 Features of visualization tools that enhance top-down comprehension (Continue)

	Data Extract Format	UML Diagrams Support	License
Hy+	GXL	Yes	Commercial
Jambalaya	GXL	Yes	Open source
***Rational Rose**	GXL	Yes	Commercial

Table 4.2 Features of visualization tools that enhance top-down comprehension

- In many cases, Rational rose is used for forward engineering, and top-down comprehension

Support Hypothesis-Driven Comprehension: Support hypothesis-driven comprehension

Different Levels Of Abstraction: Different levels of abstraction

Relational Views: Relational Views, indicate interrelationship of program component, such as class diagram, sequence diagram, collaboration diagram

Usability: Usability, providing both invoice and experienced programmer easier access

Data Extract Format: Data extract format, such as GXL (Graph exchange Language), XML (Extensible Markup Language), XMI (XML Metadata Interchange)

UML Diagrams Support: UML diagrams support

License: License, such as freeware, shareware, open source, commercial

In Hy+ system, a novel visual language called *Grapglog* is used to query database. In Jambalaya, an ontology editor called Protégé is used to support knowledge acquisition. Ontology is a description of concepts and relationships and provides an agreement of a

domain understanding. In particular, Jambalaya supports for mapping domain knowledge to code and switching between mental models would be useful. Rational Rose uses UML semantics to support hypothesis driven comprehension. Comparing these two technologies, the UML semantics is closer to the program behavior and more straightforward from a typical user perspective. Hy+, Jambalaya and Rational Rose all support UML diagrams. However, Hy+ and Jambalaya only support parts of the UML notation, compared to Rational that provides support for the complete set of UML diagrams. In Jambalaya and Hy+, a better navigation method called SHriMP (Simple Hierarchical Multi-Perspective) is used that encompass meaningful orientation cues and effective presentation is used for exploring complex software structures. Rational lacks orientation cues for switching form one view to another, making its navigation and context switching less intuitive.

4.3 Visualization Tools For Opportunistic Program Comprehension

In opportunistic program comprehension, opportunistic comprehension and top-down comprehension are often is switched. Typically, some features of visualization tools that enhance opportunistic comprehension are: [SFM99]:

- Construction of multiple mental models (domain, situation, program)
- Cross-reference mental models

4.3.1 Rigi

Rigi [MWT94] is a reverse engineering system developed to extract, navigate, analyze, and document the structure of evolving software systems (to aid software maintenance

and reengineering activities). The first phase of Rigi is extraction, which is automatic and involves parsing the software and storing the extracted artifacts. Rigi has parsers for several imperative languages, including C and COBOL. The results of first phase are a flat resource-flow graph that can be manipulated using the Rigi editor. A fisheye view technique is used to represent the large pools information. As the name applies, the fisheye technique emulates the behavior of a fisheye lens. The information at the center of the view is magnified, whilst that at the periphery is reduced in size. Also nested graphs are used for the display of software structures. The nesting nodes represent the hierarchical structure of the software. The next phase is semi-automatic; a mental model of the structure of system is made on the flat graph. Rigi depends on heavily on the experience and domain knowledge of the software engineer using it. The user makes all important decisions [MWT94]. To manage the complexity of large software systems, the second phase involves pattern-recognition skills and features subsystem-composition techniques to generate multiple, layered hierarchies of high-level abstractions [MOTU93]. Rigi supports a scripting language that allows users to customize, combine, and automate reverse engineering activities in novel ways [MWT94]. Efforts are proceeding to make user interface and configuration settings more user-customizable [Tilley95]. This approach permits the analyst to tailor the environment to better suit their needs, providing a smooth transition between automatic and semi-automatic reverse engineering [TWSM94].

4.3.2 PUI

PUI (**P**rogram **U**nderstanding **I**mplement) [CM97] is a simple browsing tool that allows maintainers to recover information as they browse through the various HTML pages. PUI

tool supports program comprehension at the top-down comprehension, bottom-up comprehension, and a combination of both. The PUI tool visualizes program elements and program relations in a smooth manner. Typically, a C program may include program elements such as types, variables, expression, statements, functions and files. Program relations include the control flow relations and the calling relations. PUI tool also provides different viewpoints. Viewpoints have different emphases on the program elements and relations and so can bring various parts of the program to the maintainers' attention [CM97]. The initial viewpoints are [CM97]:

- Information display
- Listing
- Control panel

The maintainer can choose different viewpoints since different viewpoint form different level of abstraction.

4.3.3 Fujaba

Fujaba [RH98] is a freely available software and was developed at the University of Paderborn in 1998. Fujaba is a UML based CASE-tool that supports code generation from class diagrams as well as activity diagrams, statecharts and collaboration diagrams. This allows the use of UML as a kind of visual programming language for the development of full fledged applications without any manual coding. The primary goal of the Fujaba project and environment is Round Trip Engineering with UML, SDM (Story Driven Modeling), Java and Design Patterns [RH98]. That means if some developer tools (e.g. a version control system) modify the generated code and if these stick to certain

coding standards, then the Fujaba environment is able to analyze the changed code to (re) create the corresponding UML specification. Fujaba is written in Java and contains almost 700 classes. The Fjaba version under examination was 0.6.3-0.

4.3.4 Imagix 4D

Imagix 4D [Imagix4D] is a reverse engineering tool that now features software documentation capabilities. The product offered by Imagix Corporation (USA) provides software developers with high-performance program understanding tools for C and C++. Using information located in its database and in the designer's source code, it documents each software component. This includes all files, classes, functions, and variables. Imagix 4D features a control flow analysis tool that analyzes the sequences and conditions of function calls and variable usages in the code. Developers also have control over the representation of the data, choosing where to insert cross-references and chart graphics into the documentation. The new release of the software development tool Image 4D constitutes an interesting example for the application of knowledge-based exploration and information visualization technologies. It helps to reverse engineer and document software that is complex, large, or unfamiliar. Knowledge-based graphics, as they are employed in Imagix 4D, provides a broad range of linked and synchronized displays, that go far beyond traditional call graphs and class hierarchies.

4.3.5 Features of Visualization Tools Enhancing Opportunistic Comprehension

The strategy of opportunistic comprehension requires that the programmer mix bottom-up theory and top-down theory during the program comprehension when it is necessary.

Thus, opportunistic comprehension has both features of bottom-up and top-down. In this section, we summarize some features of visualization tools that enhance opportunistic comprehension. A comparison table is shown below.

	Multiple Views	**Decomposition**	**Navigation**	**Support Hypothesis-Driven Comprehension**
Rigi	Yes Source code view, and higher-level graphical view	Yes A parsing subsystem	Yes Hypertext links	Yes A scripting language
PUI	Yes Source view, and HTML document	No	Yes Hypertext links	Yes Different viewpoints link to the corresponding implement
Fujaba	Yes UML class diagram and behavior diagram, story SDM diagram, generated source code	No	No	Yes Design pattern
Imagix 4D	Yes Providing a broad range of links and synchronized display	No	Yes Linked display	Yes Knowledge-based graphics generation

Table 4.3 Features of visualization tools that enhance opportunistic comprehension (Continue)

	Different Levels Of Abstraction	Relational Views	Cross-Reference Multiple Mental Models	Usability
Rigi	Yes Subsystem composition	Yes Graph nodes and arcs	Integration of source code view in graphical views	Yes User interface is user-customizable
PUI	Yes viewpoints	Yes viewpoints	Yes HTML document and source code views are shown together	Yes By selecting different parameter of viewpoint, there have different contents of HTML document
Fujaba	Yes Combine with class diagram, UML behavior diagram, SDM story diagram	No	Yes Combine with class diagram, UML behavior diagram, SDM story diagram	No
Imagix 4D	Yes A control flow analysis	Yes Graph nodes and arcs	Yes Looking at information database, and source code	No

Table 4.3 Features of visualization tools that enhance opportunistic comprehension (Continue)

	Source Analysis	Data Extract Format	UML Diagram Support	license
Rigi	Slicing, parsing, Data flow analysis	GXL	Yes	Open source
PUI	UML diagram	GXL	Yes	Commercial
Fujaba	UML diagram, SDM	GXL	Yes	Open source
Imagix 4D	Querying , control analysis tool	GXL	No	Shareware

Table 4.3 Features of visualization tools that enhance opportunistic comprehension

Multiple Views: Multiple Views, source code view and graphical view

Decomposition: Decomposition, a complex system can be decomposed into subsystem

Navigation: Navigation, reducing the effects of decomposition

Support Hypothesis-Driven Comprehension: Support hypothesis-driven comprehension

Different Levels Of Abstraction: Different levels of abstraction

Relational Views: Relational Views, indicate interrelationship of program component, such as class diagram, sequence diagram, collaboration diagram

Cross-Reference Multiple Mental Models: Cross-references multiple mental models

Usability: Usability, providing both novice and experienced programmer easier access

Source Analysis: Source analysis

Data Extract Format: Data extract format, such as GXL (Graph exchange Language), XML (Extensible Markup Language), XMI (XML Metadata Interchange)

UML Diagrams Support: UML diagrams support

License: License, such as freeware, shareware, open source, commercial

Tools enhancing opportunistic comprehension have both the features of enhancing bottom-up software comprehension and top-down software comprehension. Comparing the four tools included in this survey, Rigi and Imagix 4D provide immediate and visible access to the lowest units such as the program code. The relational views indicating the interrelationship of the program component are presented in the form of a graph where nodes represent software objects and arcs show the relations between the objects. PUI uses viewpoints to bring various parts of the program to the users' attention. One key feature of tools enhancing top-down software comprehension is the support of hypothesis driven comprehension. A scripting language as provided by Rigi, is used to record the hypotheses in novel ways for future maintenance. This approach allows user to tailor the environment to better suit their needs. The approach provided by Rigi, supporting hypothesis-driven is more customizable comparing with the three of others. As a tool enhancing opportunistic comprehension, Rigi has many advantages towards an effective interface for software visualization. The customizability, and flexibility of Rigi's scripting language is one of the main reasons, for its wide-spread use (at least in terms of comprehension tools). Rigi, at the current state-of-the-art is the comprehension tool, other tools have to compare and compete with. Nevertheless, this is not say that Rigi is perfect.

Rigi is missing support for advance visualization techniques (2/3D), layout algorithms are limited and the integration of source code analysis techniques could be further enhanced.

5. Summary

This paper presents a categorization for software visualization. This categorization entails and describes some important concepts. Firstly, program comprehension is an important issue in software maintenance. However, program comprehension is not solely a maintenance issue, and it is also a significant task during implementation, testing and debugging. There are four accepted categories of theories that describe the cognitive processes involved in program comprehension: bottom-up program comprehension, top-down program comprehension, opportunistic program comprehension, and an integrated meta-model of program comprehension. Several models are discussed and compared. The evaluated models are Shneiderman's model, Pennington's model, Brooks' model, Soloway and Ehrlich's model.

Software visualization is a subset of information visualization. Through software visualization, computer graphics are built to represent program comprehension. The related visualization techniques are used to build visualization tools. Supporting the user in the performance of program comprehension tasks is a major goal of visualization tools.

The contribution of this paper is the review and category of software visualization tools. The report categories the tools based on their supported comprehension strategies: top-down, bottom-up and opportunistic.

To conclude, we believe program comprehension is a difficult and important issues in software engineering. Software visualization is an effective way to support the program comprehension tasks. Support should be provided to aid the user in the construction of mental models, and the performance of program comprehension strategies.

References

[BE94] Baker, M.J., and Eick, S.G. (1994) Space Filling Software Visualization, Journal Languages and Computing, Vol.6, pp 119-133, 1994.

[Boe81] Boehm, B.W. (1981) Software Enginerring Economics. Prentice Hall, 1981.

[BOO91] Booth, G. (1991) Object-Oriented design with applications. Benjamin/Cummings Inc, California,1991.

[Bro77] Brooks, R. (1977) Towards a theory of the Cognitive Processes in Computer Programming. International Journal of Man-Machine Stidies,9(6):737-741,1977.

[Bro83] Brooks, R. (1983) Towards a theory of the Cognitive Processes in Computer Programming. International Journal of Man-Machine Studies, 18:543-554,1983.0

[BRJ99] Brooch, G., Rumbaugh, J., and Jacobson, I. (1999) The Unified Modeling Language User Guide], by Grady Booch, James Rumbaugh, Ivar Jacobson, 1999 Addison Wesely.

[CC90] Chkofsky, E.J., Cross, J.H., (1990) Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software,7(1), pp.13-17,Jan.1990.

[CM97] Chan, P-S., Munro, M. (1997) PUI: A Tool to Support Program Understanding. Published in the 5th International Workshop on Program Comprehension 1997(IWPC'97). May 28-30, 1997 in Dearborn, Michigan,USA.

[Cog82] Coggins, J.M. (1982) A framework for Texture Analysis Based on Spatial

Filtering. PhD. Thesis, Computer Science Department, Michigan State University, East Lansing, Michigan, 1982.

[Con87] Conklin, J. (1987) Hypertext : An Introduction And Survey . IEEE Computer. 20(9): 17-41. 1987.

[Coo84] Cook, R.L. (1984) Shade Trees. Proceedings of SIGGRAPH 84 (Minneapolis, Minnesota, July 23-27,1984). In Computer Graphics, v18n3. ACM SIGGRAPH, July 1984. pp223-231.

[Cor89] Corbi, T. A., (1989) Program Understanding: Challenge for the 1990s, IBM System Journal, 28(2):294-306(1989).

[Cor90] Corbi, (1990) T.A., Program Understanding: Challenge for the 1990s, IBM System Journal, Vol 28.

[CRM91] Card, S.K., Robertson., G.G., Machkinlay, J.D. (1991) The Information Visualizer: An Information Workspace Xerox Palo Alto Research Center, Palo Alto, California 94304, 1991.

[Cru01] Crumpton, J. (2001) Towards a theory of the comprehension of computer program, spring 2001.

[FOR93] Ford, L. (1993) How programmers visualize programs. Research Report271, Department of Computer Science, University of Exeter, Exeter, U.K.,1993.

[GHJV95] Gamma, E., Helm, R., Johnson, R., and Vlissides, John. (1995) Design Patterns: Elements of Resuable Object-Oriented Software. Addision-Wesley, 1995.

[GJS91] George, G.R., Jock, D.M., and Stuart, K.C. (1991) Cone Trees: 3D Animated 3D Visualizations of Hierarchical information. In Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems and graphics Interface, ACM SIGCHI, pages 189-194, New York, NY, USA 1991. ACM Press.

[Grah01] Graham, Ian. (2001) Object-Oriented Methods. Principles & Practice 3rd edition.

[GS93] Garlan, D., and Shaw, M. (1993) An introduction to Software Architecture. In Advances in Software Engineering and Knowledge Engineering (Ambriola V. Tortora G, Eds) World Sciencetific Publishing Company, 1993.

[Imagix4D] Imagix 4D . Imagix Corporation. http://www.imagix.com/index.html

[JS91] Johnson, B., and Shneiderman, B. (1991) Treemaps: a space-filling approach to the visualization of hierarchical information structures. In Proceedings of the Ieee Visualization'91, page 284-291, San Diego, CA, October,1991.

[Knight98] Knight, C. (1998) Visualisation for Program Comprehension: Information and Issues. Report 12/98.

[KS94] Kraemer, K., and Staskko, J.T. (1994) Issues in Visualization for the Comprehesion of Parallel Programs. College of computing, Georgia Institute of Technology, Atlanta, GA 30332-0280 , E-mail:{ stasko,Eileen}@cc.gatech.edu. 1994.

[LA94] Leung, Y., Apperley, M, (1994) A Review and Taxonomy of Distortion-

Orientation Presentation Techniques, ACM Transaction on Computer = Human

interaction 1 (2), 1994, 126–160.

[LCCCLY95] Lu, C.W., Chu, W.C., Chang, C.H., Chung, Y.C., Liu, X., and Yang, Y. (1995) Reverse Engineering. Department of Information Engineering, Tunghai University, Taichung, Taiwan, chu@ csie.thu.edu.tw. School of Computing , Napier University, Edinburgh, Scotland. Department of Computer Science , De Montfort University, Leicester, England.

[Let86] Letovsky, S. (1986) Cognitive processes in program comprehension. In Empirical studies of programmers, pages 58-79. Ablex Publishing Corporation,1986.

[LH92] Levkowitz, H., and Herman, G.T. (1992) Color scales for image data. IEEE Computer Graphics and Applications, 12(1): 78-80, 1992.

[LHMR92] Levkowitz, H., Holub, R.A., Meyer, G.W., and Roberson, P.K. (1992) Color vs black and white in visualization. IEEE Computer Graphics and Applications, 12(4): 20-22, 1992.

[LM94] Leung, Y.K., Apperley, Mark D.(1994) A Review and Taxonomy of Distortion-Oriented Presentation Techniques. ACM Transactions on Computer-Human Interaction, 1(2):126-160, June 1994.

[LR94] Lamping, J., Rao, R. (1994) Laying out and Visualizing Large Trees Using a Hyperbolic Space. Nov 2-4, 1994.

[LRP95] Lamping, J., Rao, R., Pirolli, P. (1995) A Focus + Context Technique Based on Hyperbolic Geometry for Visualizing Large Hierarchies. Proceedings of ACM CHI

1995,401-408.

[LS94] Livadas, P. E., and Small, D. T., (1994) Understanding Code Containing Preprocessor Constructs. In: Proceedings of the 3rd Workshop on Program Comprehension, Washington, DC, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp.89-97.

[LW86] Lyle, J., and Weiser, M. (1986) Experiments on slicing-based debugging tools. Proc. Of the 1th Conf. On Empirilcal Studies of Programming, pp.187-197,6/1986.

[MBCCK93] MORRISON, R., BAKER, C., CONNOR, R.C.H., CUTTS, Q.I, and KIRBY, G.N.C. (1993) Approaching Integration in Software Environments. Department of Mathematical and Computational Sciences, University of St Andrews, North haugh, St Andrews, 1993.

[MFGNCG00] Musen, M.A., Fergerson, R.W., Grosso, W.E., Noy, N.F., Crubezy, M., and Gennari, J.H. (2000) Component-based support for building knowledge-acquisition systems. Proceedings of the Conference on Intelligent Information Processing (IIP 2000) of the international Federation for Information Processing Sixteenth World Computer Congress (WCC 2000), Beijing, China, August, 2000, pp.18-22.

[Microsoft97] Microsoft Corporation. Redmond – WA. Internet Explorer, November 1997. Online http:// www.microsoft.com

[MOTU93] Muller, H.A., Orgun, M.A., Tilley, S.R., and Uhl, J.S. (1993) A reverse engineering approach to subsystem structure identification. Journal of Software

Maintenance: Research and Practice, 5(4):181-204, December 1993.

[MS95] Mendelzon, A., and Sametinger, J. (1995) Reverse engineering by visualizing and querying. Software- Concepts and Tools, 16:170-182, 1995.

[MS02] Matteo, C., Sergio, m. (2002) Information Visualization and Usage a Perspective. June 14,2002.

[MUN71] Munsell, A.H. (1971) A color Notation (12^{th} Edition). Munsell Color Company, Baltimore, Maryland, 1971.

[MV93a] Mayrhauser, A.V., and Vans, A.M. (1993) Comprehension Process During Large Scale Maintenance. *Research Paper.* Dept. of Computer Science Colorado State University Fort Collins, CO 80523.1993.

[MV93b] Mayrhauser, A.V., and Vans, A.M. (1993) From Program comprehension to Tool Requirements for an Industrial . In: *Proceedings of the 2^{nd} Workshop on Program Comprehension, Capri, Italy, pp, 78-86, July 1993.*

[MV93c] Mayrhauser, A.V., and Vans, A.M. (1993) From Code Understanding Needs to Reverse Engineering Tool Capabilities, In: *Proceedings of the 6^{th} International Workshop on Computer-Aided Software Engineering (CASE93), Singapore, pp:230-239,July 1993.*

[MV94] Mayrhauser, A.V. and Vans, A.M. (1994) Program Understanding- A Survey. Technical Report CS-94-120, August 23, 1994. Colorado State University.

[MV95] Mayrhauser, A.V. and Vans, A.M. (1995) Program comprehension during software maintenance and evolution. IEEE Computer, pages 44-45, August 1995.

[MWT94] Muller, H.A., Wong, K., Tilley, S.R. (1994) Understanding Software Systems Using Reverse Engineering Technology. Department of Computer Science, University of Victoria. P.O. Box 3055, Victoria, BC, Canada V8W 3P6. 1994.

[MYE90] Myers, B A.(1990) Taxonomies of visual programming and program visualization. Journal of Visual Languages and Computing, 1(1): 97-123,1990.

[Netscape97] Netscape Communications. Mountain View. CA. Netscape Navigator. November 1997. Online http:// www.netscape.com

[Nie90] Nielsen, J. (1990) The Art Of Navigation Through Hypertext. Communications of the ACM. 33(3): 296-310, may 1990.

[Nie95] Nielsen, J. (1995) Multimedia And Hypertext: The Internet And Beyond. Academic Press Inc., 1995.

[NS84] NAGY,G., AND SETH, S. (1984) Hierarchical representation of optically scanned documents". In Proceedings of the IEEE 7th International Conference on Pattern Recognition (Montreal, Canada,1984, pp.347-349.

[NSDCFM01] Noy, F.N., Sintek, M., Decker, S., Crubezy, M., Fergerson, R.W., and Musen, M.A. Creating Semantic Wen contents with Protégé- 2000. IEEE Intelligent Systems, 16(2): 60-71, 2001.

[OL98] Olano, M., Lastra, M. (1998) A Shading Language on Graphics Hardware: The Pixelflow Shading System. Proceedings of SIGGRAPH 98, Orlando, Florida, July 19-24,1998.

[Oman90] Oman, P. Maintenance tools. IEEE software, 7 (3): 59-65, May, 1990.

[Pen87] Pennington, N. (1987) Stimulus structures and mental representations in expert comprehension of computer programs. Cognitive psychology,19:295-341,1987.

[Protégé] Protégé project, Stanford University, http://protégé.stanford.edu.

[Rational] http://www.rational.com/

[RA96] Rajlich, V., Adnapally, R. (1996) VIFOR 2: A Tool for Browsing and Documentation, IEEE International Conference of Software Maintenance, pp 296-300, 1996.

[RBCM91] Robson, D.J., Bennett, K.H., Cornelius, B.J., and Munro, M. (1991) Approaches to Program Comprehension. Journal of Systems and Software 14 (1991), 79-84.

[RC92] Roman, G.-C., Cox, K. C. (1992) *Program Visualization: The Art of Mapping Programs to Pictures*. Proc. of the International Conference on Software Engineering, Association of Computing Machinery, 1992.

[RDL90] Rajlich, V., Damaskinos, N., and Linos, P. (1990) VIFOR: A Tool for software Maintenance, Software Practice and Experience, Vol.20, No.1, pp67-77, January 1990.

[RF00] RoBling, G., Freisleben, B. (2000) Program Visualization Using ANIMALSCRIPT.

[RGD94] Rajlich, V., Gudla, R., Doran, J. (1994) Layered Explanations of Software: A Methodology for Program Comprehension, In: 1994 IEEE Workshop on Program

Comprehension, 46-53.

[RH98] Rockel, I., and Heimes, F. (1998) Fujaba-Homepage.

http://www.uni-paderborn.de/cs/fujaba/

[RSF00] RoBling,G., Schuler, M., Freisleben, B. (2000) The ANIMAL Algorithm Animation Tool.

[Rug81] Rugaber, S. (1981) Program Comprehension. Georgia Institute of Technology, May 4, 1995.

[SAE88] Soloway, E., Adelson, B., and Ehrlich, E. (1988) Knowledge and Processes in the comprehension of Computer Programs, In: The nature of Expertise , Eds. M. Chi, R. Glaser, and M.Farr, 1988, Alawrence Erlbaum Associates, Publishers, pp.129-152.

[Sam89] SAMET, H. (1989) Design and Analysis of Spatial Data Structures. Addision-Wesley Publishing Co., Reading, MA,1989.

[SE84] Soloway, E., and Ehrlich, K. (1984) Empirical Studies of Programming Knowledge, In: IEEE Transaction on Software Engineering, September 1984, Vol. SE, No. 5, pp. 595-609.

[SFM99] Storey, M.-A.D., Fracchia, F.D., Muller, H.A., 1999) Cognitive Design Elements to Support the Construction of a Mental Model during Software Visualization. School of Computing Science Simon Fraser University, Burnaby, BC , Canada, Department of Computer Science , University of Victoria, BC, Canada. 1999.

[SHH98] Smith, R., Hixon, R., Horan, B. (1998) Supporting Flexible Roles in a Shared

Space, Proceedings of ACM CSCW'98, 197 – 206.

[Shn80] Shneiderman, B. (1980) Software Psychology: Human Factors in Computer and Information Systems. Winthrop Publishers, Inc.,1980.

[Shn91] Shneidreman, B. (1991) Tree Visualization with Tree-Maps: 2-d Space-Filing Approach. University of Maryland, 1991.

[ShriMP] ShriMP Views project, University of Victoria,

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

[SK93] Stasko, J. T., Kraemer, E.. (1993) A methodology for building application-specific visualizations of parallel programs. Joural of Parallel and Distributed Computing, 18(2): 258-264, June `993.

[Skl78] Sklansky, J. (1978) Image Segmentation and Feature Extraction. IEEE Transactions on Systems, Man and Cybernetics, SMC-8, pp.237-247,1978.

[SM79] Shneiderman, B., and Mayer, R. (1979) Syntactic/ Semantic Interactions iProgramer Bahavior: A Model and Experimental Results, In: International Journal of Computer and Information Sciences, 1979,Vol.8, No.3,pg.219-238.

[SM00] Staab, S., and Maedche, A. (2000) Ontology Engineering beyond the Modeling of Concept and Relations.

[SMW95] Storey, M-A.D., Muller, H.A., Wong, W. (1995) Manipulating and Documenting Software Structures Using ShriMP Views, proceedings of ICSM'95, pp 275-284, October 17-20,1995.

[Som95] Sommerville, I. (1995) *Software Engineering*. Fifth edition, Addison-Wesley Publishing Co. Inc., Wockingham, England,pp.700-712,1995.

[SMSBEFN01] Storey, M-A., Musen, M., Silva, J., Best, C., Ernst N., Fergerson, R., Noy, N. (2001) Jambalaya: Interactive visualization to enhance ontology authoring and knowledge acquisition in Protégé.

[ST78] Swanson, B.L.E., and Tompkins, G.E. (1978) Characteristics of Application Software Maintenance. Communications of the ACM, 21(6), June 1978.

[Sta84] Standish, T. A., (1984) An Essay on Software Reuses. IEEE Transactions on Software Engineering, SE-10(5): 494-497(1984).

[SWFM97] Storey, M-A.D., Wong, K., Fracchia, F.D., Muller, M.A. (1997) On Integrating Visualization Techniques for Effective Software Exploration.

[TC90] Tyler, C.W. and Clarke, M.B. (1990) The autostereogram. SPIE Stereoscopic Display and Applications 1258: 182-196.

[Tilley95] Tilley, S.R. (1995) Domain-retargetable reverse engineering II: Personalized user interfaces. In international Conference on Software Maintenance (Icsm'94), (Victoria, BC; September 19-23, 1994), page 366-342. IEEE Computer Society Press (Order Number 6330-02), September 1994.

[TJ98] Tuceryan, M., and Jain, A.J. (1998) Texture Analysis.

[TMY78] Tamura,H., Mori, S., and Yamawaki, Y. (1978) Textural Features Corresponding to Visual Perception. IEEE Transactions on Systems, Man and

Cybernetics, SMC-8, pp.460-473, 1978.

[Tru81] Truckenbrod, T.R. (1981) Effective Use Of Color in Computer Graphics. Department of Art Northern Illinois University Dekalb, Illinois.

[Tuf90] Tufte, E. (1990) Envisioning Information, Graphics Press,1990.

[TWSM94] Tilley, S.R., Wong, K., Storey, M.-A. and Muller, H.A. (1994) Programmable reverse engineering. To appear in the International Journal of software Engineering and Knowledge Engineering, 4(4), December 1994.

[WL95] Ware, C., Lewis, M. (1995) The DragMag Image Magnifier. Video Proceedings of ACM CHI'95, 1995, vol.2, p.407-408.

Appendix

Diagrams in the UML:

Class diagram - Shows a set of classes, interfaces, and collaborations and their relationships. This is the most common type of diagram used when modeling OO systems.

Object diagram - Shows a set of objects and their relationships. Can be thought of as an instance of a Class diagram.

Use case diagram - Shows a set of use cases and actors and their relationships. These types of diagrams drive the whole development process since they describe the requirements of the system.

Sequence diagram - Shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them. Emphasizes the time ordering of messages.

Collaboration diagram - Shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them. Emphasizes the structural organization of the objects that send and receive messages.

Statechart diagram - Shows a state machine, consisting of states, transitions, events, and activities.

Activity diagram - Special kind of a Statechart diagram that shows the flow from activity to activity within a system, and they are very similar to flowchart diagrams except that concurrency may be modeled in Activity diagrams.

Component diagram - Shows the organizations and dependencies among a set of components.

Deployment diagram - Shows the configuration of run - time processing nodes and the components that live on them.

Views in UML:

Use case view – Encompass the use cases that describe the behavior of the system as seen by its end users, analysts, and testers

Design view - Encompass the classes, interfaces, and collaborations that form the vocabulary of the problem and its solution

Process view - Encompass the threads and processes that form the system's concurrency and synchronization mechanisms

Implementation view – Encompass the components and files that are used to assemble and release the physical system

Deployment view - Encompass the nodes that form the system's hardware topology on which the system executes