A Survey and Categorization of Program Comprehension Techniques

Song Wei

A Major Report

in

The Department

of

Computer Science

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

March 2002

0-612-68487-3

ABSTRACT

A Survey and Categorization of Program Comprehension Techniques

Song Wei

Program comprehension is a central activity during software maintenance, evolution and reuse. Some reports estimate that up to 60-70% of the maintenance effort is spent in trying to understand code. Poor design, unstructured programming methods, and crisis-driven maintenance can contribute to poor quality code, which in turn affects program comprehension. The implications are that improvements to software development process will require improvements to software maintenance. These process improvements should facilitate comprehension of existing programs. The goal of program comprehension is to acquire sufficient knowledge about a software system so that it can evolve in a disciplined manner. Program comprehension is an emerging interest area within the software engineering field. In this report, the objective is to survey and categorize program comprehension techniques. We also present the MOOSE project to provide an example to illustrate some of these comprehension survey and categorization.

Acknowledgement

First of all, I would like to thank my supervisor, Dr. Juergen Rilling for his encouragement and valuable suggestions. Without his help, the work would not be what it is today.

Second, I would also like to thank Dr. Sabine Bergler, give me many comments for the thesis.

Finally, I would like to dedicate the work to my family, for their support and encouragement during my master program study.

Table of Contents

List of Figures

1 Introduction

Program comprehension is a central activity during software maintenance, evolution, and reuse. Some reports estimate that up to 60-70% of the maintenance effort is spent in trying to understand code. Poor design, unstructured programming methods, and crisis-driven maintenance can contribute to poor quality code, which in turn affects program comprehension. The goal of program comprehension is to acquire sufficient knowledge about a software system so that it can evolve in a disciplined manner. There are varieties of support mechanisms for aiding program comprehension, which can be grouped into three categories: unaided browsing, leveraging corporate knowledge and experience, and computer-aided techniques like reverse engineering. During the last three decades, the human computer interaction community has addressed this question by developing different techniques and approaches to validate comprehension and visualization techniques with users. It is essential to derive techniques and algorithms that provide "means" and insights to the data to be displayed.

In this report, we focus on a survey and categorization of program comprehension techniques. One approach to improve the comprehension of software systems is through reverse engineering (bottom up) by providing higher level of abstraction through visualizing of data that has to be observed and inspected. However, no matter what visualization technique we choose, the sheer volume of information presented to the developers becomes daunting, as programs grow more complex and large. Another approach to improve the comprehension of software systems is through forward engineering (top down) by using formal specification, use case, and use case maps etc. techniques.

1.1 Motivation and Objective

Program comprehension is an emerging interest area within the software engineering field. Software engineering itself 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 on maintaining existing systems rather than developing new ones [Rug95]. Traditionally, over 70% of the total expenditure for software is directed to software maintenance and about 60-70% of this expenditure is directed towards understanding the system. The implications are that if we want to improve software development, we should look at maintenance, and if we want to improve maintenance, we should facilitate the process of comprehending existing programs.

In this report, the objective is to survey and categorize program comprehension techniques. We present the MOOSE project to provide an example to illustrate some of these comprehension techniques surveyed and categorized.

1.2 Scope of Thesis

The presented thesis consists of five sections including this section. In section two, an overview of program comprehension and a general introduction on cognitive models is presented. In the third section, a survey of program comprehension techniques is presented. We introduce and categorize several techniques for visualization, algorithmic and application support. In section four, the

MOOSE project and its task driven approach is presented. In the last section, conclusions and future directions related to this research are presented.

2 Program Comprehension & Cognitive Models

In this section, we will discuss background, need and some problem for program comprehension and introducing four different cognitive models and their applications.

2.1 Program comprehension background

Program comprehension is a crucial part of system development and software maintenance. It is expected that a major share of systems development effort goes into modifying and extending preexisting systems, about which we usually know little [Gal91]. As Mayrhauser mentioned in her paper [May98]: "software maintenance of existing systems consumes 50–70% of the total programming effort and a significant portion of this maintenance activity (50-70%) is spent on software understanding. Because of the increased complexity of software systems, their maintenance is becoming more and more aggravating. Change to a system may be necessitated for adaptive, perfect, corrective or preventive reasons. Understanding the system, incorporating the change, and testing to ensure that the change has no unexpected effect on the system are the three facets of software maintenance."

Cognitive models are used to describe a maintainer's mental representation of the program to be understood. And it describes the information structures used to form the mental model [Sto98]. In what follows some key cognitive models will be discussed.

2.2 The need for Program Comprehension

Spencer Rugaber described [Rug95] program comprehension as the process of acquiring knowledge about a computer program. Increased knowledge enables such activities as bug correction, enhancement, reuse, and documentation. While efforts are underway to automate the understanding process, such significant amounts of knowledge and analytical powers are required that today program comprehension is still largely a manual task.

Program comprehension is a gradual process of building up the necessary understanding by examining sections of the source code. Using the knowledge gained from the source code, explanations and understanding of the software system under investigation can be built and refined.

Also Spencer Rugaber mentioned in [Rug95]: "Program comprehension is an emerging interesting area within the software engineering field. Software engineering itself 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 on maintaining existing systems rather than developing new ones. So for the software maintenance process, the greater part is devoted to understanding the system being maintained. Fjeldstad and Hamlen report that 50% to 70% of time spent on actual enhancement and correction tasks, respectively, are devoted to comprehension activities. These involve reading the documentation, scanning the source code, and understanding the changes to be made [Fje83].

2.3 Problems and limitations of Program comprehension

Program comprehension is a research area, which belongs to the software engineering domain and is devoted to developing tools and methodologies to aid in the understanding and management of the increasing number of legacy systems. As Spencer Rugaber described [Rug95]: "software engineering itself 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 on maintaining existing system rather than developing new ones. According to Barry W. Boehm described in Software Engineering Economics1981 [Boe81]: estimates of the proportion of resource and time devoted to maintaining legacy systems range from 50% to 70% in the total software budget."

Current software systems are difficult to comprehend because their size and complexity far exceeds that of the human brain but each brain are different. It is this difference that requires research into easing program understanding to provide alternative ways for discovering information and validating hypotheses. According to Spencer Rugaber mentioned in [Rug95]: "Program comprehension is difficult because it must bridge different conceptual areas. Of particular importance are bridges over the following five gaps."

- 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.

2.4 Cognitive Models

A cognitive model describes a maintainer's mental representation of the program to be understood. A cognitive model describes the cognitive processes and the information structures used to form mental models. Over the past 20 years, researchers have already proposed many studies to observe how programmers understand programs. They developed a variety of support mechanism to aiding program comprehension or understanding.

Scott Tilly [Til98] described in "A reverse Engineering Environment Framework", they can be classified into three categories:

(a) Unaided browsing;

(b) Leveraging corporate knowledge and experience (Top Down Model);

(c) And computer-aided techniques like reverse engineering (Bottom Up).

Mayrhauser and Vans mentioned in [May92]: "Studies have shown that, in reality, software engineers switch between these different models depending on the problem-solving task. They proposed an *opportunistic* or integrated approach, which is combining both top-down and bottom-up cues as they become available.

2.4.1 Unaided browsing

Unaided browsing is actually "human ware", which is software maintainer manually look through source code or software document in printed form or browses it online, perhaps using the file system as a navigation aid [Til98]. This method is a widely applied approach for the comprehension of large systems, and frequently used by users in some form. Maybe a good software engineer can keep track of approximately 40-50,000 lines of code in his/her head, for larger programs however, it is impossible to keep track of all information. Additional tools and techniques have to be applied to support the comprehension process for larger programs.

2.4.2 Leveraging corporate knowledge and experience (Top Down Model)

Brooks [Bro83] theorizes that programmers understand a complete program in a top-down manner, where the comprehension process is one of reconstructing knowledge about the domain of the program and mapping that to the actual code itself. The process starts with a hypothesis about the goal nature of the program. The initial hypothesis is then refined in a hierarchical fashion by foaming subsidiary hypotheses. Subsidiary hypotheses are refined and evaluated in a depth first manner to reduce the cognitive load of the programmer. The verification (or rejection) of hypotheses depends heavily on the absence or presence of *beacons*. A beacon is a set of features that indicates the existence of hypothesized structures or operations. An example of a beacon may be a function called swap in a program. The discovery of a beacon permits code features to be bound to hypotheses.

Similarly, Soloway and Ehrlich [Sol84] observed that top-down comprehension is used when the code or type of code is familiar. They observed that expert programmers use two types of programming knowledge:

- *Programming plans* are generic fragments of code that represent typical scenarios in programming. For examples, a sorting program will contain a loop which compares two numbers in each iteration.
- *Rules of programming discourse* capture the conventions of programming, such as coding standards and algorithm implementations.

Based on Soloway and Ehrlich's observations, a mental model is formed top-down by forming a hierarchy of goals and programming plans. Rules of discourse and beacons help decompose goals and plans into lower level plans.

2.4.3 Computer-aided Techniques (Bottom Up Model)

As G.A. Miller mentioned in Psychological Review [Mil56]: Experimentation has shown that there is a limitation on the number of separate pieces of information that can be stored in a person's short term memory at any one time. The phrase *chunking* describes the process of recoding information into groups so that more information can be stored in short-term memory. The bottom-up theory of program comprehension is related in that it states that programmers first read statements in the code and then mentally chunk or group these statements into higher level abstractions. Shneiderman [Shn79] proposed that programs are understood bottom-up, by reading the source code and then mentally chunking the low-level software artifacts into meaningful, higher-level abstractions. These abstractions are further grouped until a high-level understanding of the program is formed.

As Scott Tilley described [Til98], two common approaches to program understanding often cited in the literature are a functional approach that emphasizes cognition by what the system does and a behavioral approach that emphasizes how the system works. These two approaches are directly related to the level of domain expertise of the software engineer. The functional approach is bottom up and deductive, relying more on the knowledge of the implementation domain to create more abstract concepts that may map to the application domain and the systems functional requirements. The bottom up approach reconstructs the high level design of a system, starting with source code, through a series of chunking and concept-assignment steps.

Pennington [Pen87] also uses bottom-up approaches for program understanding. She identifies that the first mental representation programmer's build is a control flow abstraction of the program called the program model. This model is built from the bottom up using beacons to identify elementary blocks of code that control primes in the program. When the program model representation exists, a situation model is developed. This model is also built from the bottom up, using a data-flow/functional abstraction. The development of the situation model requires knowledge of the application domain.

2.4.4 Integrated or Opportunistic Approach

Both top-down and bottom-up comprehension models have been used in an attempt to define how a software engineer understands a program. However, case studies have shown that, in industry, maintainers of large-scale programs frequently switch between these different models depending on the problem-solving task at hand [May92]. The opportunistic approach involves creating, verifying, and modifying hypotheses until the entire system can be explained using a consistent set of hypotheses.

The integrated Meta-Model, developed by von Mayrhauser and Vans, consists of four major components. The first three components describe the comprehension processes used to create mental representations at various levels of abstraction and the fourth component describes the knowledge base needed to perform a comprehension process.

The integrated model combines the top-down understanding with the bottom-up understanding, recognizing that for large systems a combination of approaches to understanding becomes necessary. Experiments showed that programmers switch between all three comprehension models [May94].

3 A Program Comprehension Framework

What is a program comprehension framework?

One of the key reasons of program comprehension or understanding is to maintain and reengineer legacy systems. It represents an objective rather than a well-defined process, and involves inverse domain mapping, that can be aided by reverse engineering. A reverse engineering environment must make the inverse process easier by recovering lost information and making implicit information explicit [Til96a].

A comprehension framework is a construction/architecture for computer aided reverse engineering support of program comprehension/understanding. A framework provides means to classify different approaches to reverse engineering and enables a common frame of reference and implementation of different tools. It should be emphasized that a framework enables the integration of different tools-it is not meant to be used as an evaluation mechanism.

Reverse engineering as part of program comprehension can be described as analyzing a subject system [Ril01]:

1) To identify the system's components and their interrelationship;
2) To create representations of a system in another form at a high level of abstraction;
3) To understand the program execution and the sequence in which it occurred.

A framework essentially implements a generic architecture for an application domain in terms of classes [Bec94]. Previous knowledge about the application domain is, doubtless, of great importance to help with a given comprehension framework. Through general domain knowledge, a programmer is able to comprehend the general organization of concepts or, more specifically, the domain model implemented by the framework. On the other hand, it's also necessary to take into account that the goal of a framework development is to allow framework users to reuse the designer domain knowledge. Therefore, it's reasonable to expect that framework users don't have a deep knowledge about the application domain, but just the essential knowledge about the functionality of the application to be implemented. Ideally, a framework should allow the user to implement applications knowing just the functionality that abstract classes leave to implemented by subclasses [Cam96].

A reasonable step in a framework comprehension process is to provide users with the mechanisms that allow them to build an initial mental model of the structure and the behavior of the architecture implemented by the framework. According to this, providing support for recognizing abstractions not supported by the programming language, is an important complement to facilitate the global comprehension of the functionality of a framework [Cam96].

As an example for these comprehension techniques, the MOOSE (Montreal Object-Oriented Slicing Environment) is used to illustrate the tool support for various cognitive models during the comprehension of large software systems. The MOOSE framework architecture is based on three major components as shown in Figure 1. These components are: (1) Software visualization providing higher levels of abstraction; (2) A hybrid slicing framework providing algorithmic support to allow for a reduction of the software complexity; (3) Application framework that combines and utilizes both algorithmic techniques and the various software visualization approaches.

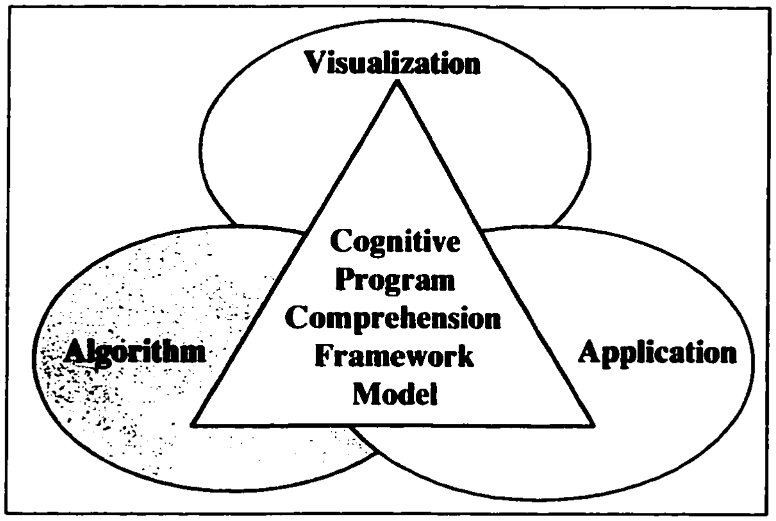


Figure 1. The three pillars of the cognitive program comprehension framework

In what follows, we present our survey and categorization of program comprehension techniques which is organized as shown below:

Visualization Techniques (3.1)	***Reading Techniques*** ***Use Case Maps (UCMs)*** ***Use Case***	***MOOSE*** ***(Sequence diagram etc)***
Algorithmic Support (3.2)	***Formal Specification*** ***Use Case Maps (UCMs)***	***MOOSE (Slicing)***
Application Support (3.3)	***Forward Engineering***	***MOOSE (Debugging etc.)***

3.1 Visualization Techniques

What is Software visualization?

Software visualization can be seen as a specialized subset of information visualization. This is because information visualization is the process of creating a graphical representation of abstract, generally non-numerical data. This is exactly what is required when trying to visualize software. The term software visualization has many meanings depending on the author. Software visualization can be defined, as "Software visualization is a discipline that makes use of various forms of imagery to provide insight and understanding and to reduce complexity of the existing software system under consideration." [Cla98]

The goal of software visualization is also included in the above definition. To create visualization for no real purpose would be a pointless exercise. It has long been known that understanding software is a complex and hard task because of the complexity of the software itself. Therefore techniques that aid programmers in his comprehension of an existing software system are at the focus of various research activities. Software visualization aims to aid the programmer by providing insight and understanding through graphical displays and views, and to reduce the perceived complexity through the use of suitable abstractions and metaphors [Cla98].

Myers [Mye90] effectively sums up the benefits of using graphics in the presentation of program information when he writes:

"The human visual system and human visual information processing are clearly optimized for multi-dimensional data. Computer programs, however, are conventionally presented in a one dimensional textual form, not utilizing the full power of the brain."

Why Visualization?

- The human brain possesses a "narrow bandwidth" for processing raw numbers, but a surprisingly "wide bandwidth" for processing visual data.
- Source code comprehension benefits from the use of an appropriate graphical notation.
- Graphical representations have been recognized as having an important impact in communicating from the perspective of both writers and readers.

3.1.1 Top down

The top-down approach

Scott Tilley described in [Til98], the behavioral approach as top down and inductive, using a goal-driven method of hypothesis postulation and refinement based on expected artifacts derived from knowledge of the application domain. The top-down approach begins with a pre-existing notion of the functionality of the system and proceeds to earmark individual components of the system responsible for specific tasks.

In what follows, we describe techniques that are adopted for the top-down approach: Reading technique, Use Case Maps (UCM) and Use Case.

3.1.1.1 Reading Techniques

Reading techniques can be applied as part of a top down visualization framework. As Basili described in [Bas95]: Software reading is a key technical activity that aims at achieving whatever degree of understanding that is needed to accomplish a particular objective. The various work documents associated with software development (e.g., requirements, design, code, and test plans) often require continual understanding, review and modification throughout the development life cycle. Thus software reading, i.e., the individual analysis of textual software work products, is the core activity in many software engineering tasks: verification and validation, maintenance, evolution, and reuse.

According to Basili's theory, the taxonomy of reading techniques is shown in Figure 2. The upper part of the tree (over the dashed horizontal line) models the problems that can be addressed by reading. Each level represents a further specialization of the problem according to classification attributes which are shown in the rightmost column of the figure. For example, reading (*technology*) can be applied for analysis (*high level goal*), more specifically to detect faults (*specific goal*) in a requirements specification (*document*) which are written in English (*notation/form*).

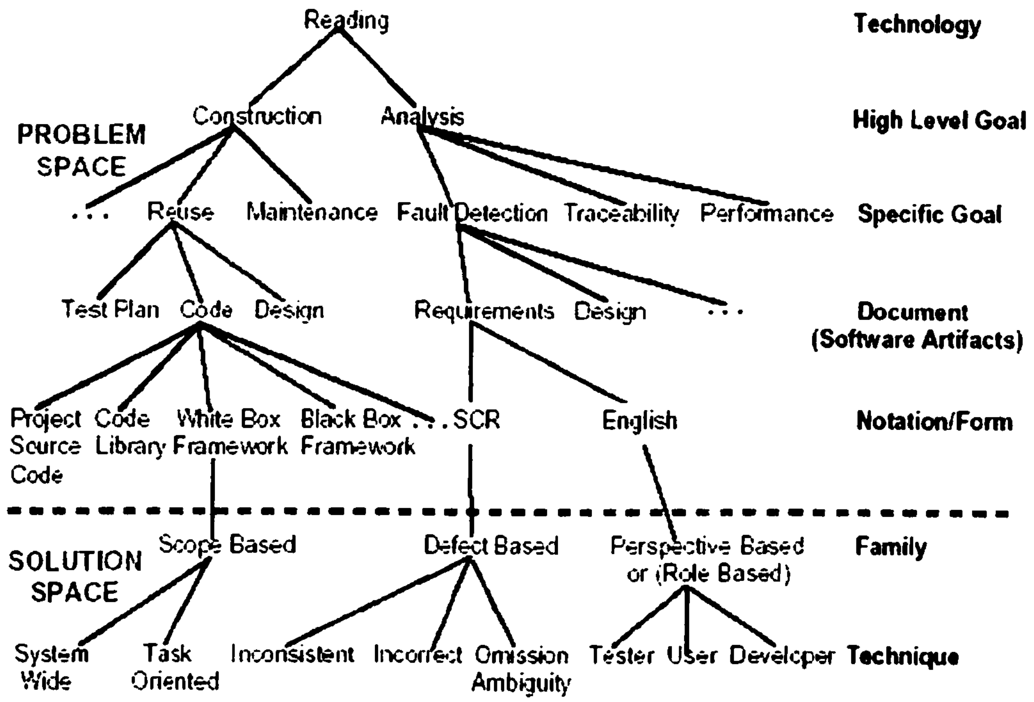


Figure 2. Families of reading techniques

The lower part of the tree, (below the dashed horizontal line) models the specific solutions we have provided to date for the particular problems, represented by each path down the tree. The solution space consists of reading families (components of reading techniques) and reading techniques. Each family is associated with a particular goal, document or software artifact, and notation in which the document is written. Each technique within the family is: (1) tailorable, based upon the project and environment characteristics; (2) detailed, in that it provides the reader with a well-defined set of steps to follow; (3) specific, in that the reader has a particular purpose or goal for reading the document and the procedures that support the goal; (4) focused, in that it provides a particular coverage of the document, and a combination of techniques in the family provides coverage of the entire document. Finally each technique is being studied empirically to determine if and when it is most effective.

Analysis Reading

Analysis reading can solve these problems: Given a document, how can we assess various qualities and characteristics? Reading for analysis is important for product quality; it can help us understand the types of defects we make in programming, and the nature and structure of the product. It can be used for various documents throughout the life cycle. Besides helping us assess quality, it can provide insights into better development techniques.

The first family of scenario-based reading techniques is known as *defect-based reading*, and focuses on a model of the requirements using a state machine notation. The different model views are based upon focusing on specific defect classes: data type inconsistency, incorrect functions, and ambiguity or missing information. The analysis questions are generated by combining and abstracting a set of questions that are used in checklists for evaluating the correctness and reliability of requirements documents.

The second family of techniques, *perspective-based reading*, focuses on different product perspectives, e.g., reading from the perspective of the software designer, the tester, the end-user, the maintainer, the hardware engineer. The analysis questions are generated by focusing predominantly on various types of requirements errors (e.g., incorrect fact, omission, ambiguity, and inconsistency) by developing questions that can be used to discover those errors from the one perspective assumed by the reader of the document (e.g., the questions for the tester perspective lead the reader to discover those requirement errors that could be found by testing the final product).

Victor Basili mentioned in his paper [Bas95]: "In order to understand the effectiveness of scenario-based reading techniques in particular, we have experimentally studied techniques from both families." He generated two families of reading techniques (collectively known as *scenario-based reading*), by creating operational scenarios which require the reader to first create an abstraction of the product, and then answer questions based on analyzing the abstraction with a particular emphasis or role that the reader assumes. Each reading technique in a family can be based upon a different abstraction and question set. The choice of abstraction and the types of questions may depend on the document being read, the problem history of the organization, or the goals of the organization.

The first series of experiments [Por95, Bas96] was aimed at discovering if scenario-based reading is more effective than current practices. Based upon these experiments, empirical evidence showed that scenario-based reading techniques could improve the effectiveness of reading methods. At the same time, it was noted that some scenarios were less effective than others. In what follows, two experiments are presented.

Defect-Based Reading Experiment

In the defect-based reading study, A. Porter described in [Por95], they evaluated and compared defect-based reading, ad hoc reading and checklist-based reading, with respect to their effect on fault detection effectiveness in the context of an inspection team. The study, a blocked subject-project, was replicated twice in the spring and fall of '93 using 48 graduate students at the University of Maryland. The experimental design was a partial fractional factorial design. The design was less elegant than the [Bas87] design because the comparison here is with the status quo approach (ad hoc) or with a less procedurally organized approach (checklists) so it is impossible to teach the subject a defect-based reading approach and then return to ad hoc or check list. In this case, a sort of ordering was assumed. On the first pass there were more ad hoc and check list readers. Several, but not all, were moved to defect-based reading on the second pass.

Major results were that:

- the defect-based readers performed significantly better than ad hoc and checklist readers;
- the defect-based reading procedures helped reviewers focus on specific fault classes but were no less effective at detecting other faults; and
- checklist reading was no more effective than ad hoc reading.

Perspective-Based Reading Experiment

For the perspective-based reading study, V.Basili [Bas96] evaluated and compared perspective-based reading and NASA's current reading technique with respect to their effect on fault detection effectiveness in the context of an inspection team. Three types of perspective-based reading techniques were defined and studied: tester-based, designer-based, and user-based. The study, again a blocked subject-project, was run twice in the SEL environment with NASA professionals. The design evaluated the effectiveness of perspective-based reading on both domain-specific and generic requirements documents, which had been constructed expressly so that the generic portion could be replicated in a number of different environments, while the domain-specific part could be replaced in each new environment. This would allow them to combine the generic parts from multiple studies but focus on improvement local to a particular environment. Based on feedback from the subjects and other difficulties encountered in the first run of the experiment, they were able to make changes to the experimental design that improved the second run. For example, they found it necessary to:

- Include more training sessions, to make certain that subjects were familiar with both the documents and techniques involved in the experiment;
- Allow less time for each review of the document, since subjects tended to tire in longer sessions;
- Shorten some of the documents, to reach a size that could realistically be expected to be checked in an experimental, time-constrained setting.

Major results of this experiment were that:

- both team and individual scores improved when perspective-based reading was applied to generic documents
- team scores improved when perspective-based reading was applied to NASA documents

Construction Reading

Construction reading can solve the following problem: Given an existing system, how do I understand how to use it as part of my new system? Reading for construction is important for comprehending what a system does, what capabilities exist and do not exist; it helps us abstract the important information in the system. It is useful for maintenance as well as for building new systems from reusable components and architectures.

Reusing class libraries does increase quality and productivity, but class libraries do not provide default system behavior but only functionality at a low level, and force the developer to provide the interconnections between the libraries [Bas95]. Greater benefits can be expected from reusable, domain specific architectures and components that are of sufficient size to be worth reusing. Thus, there is currently a focus on the reuse allowed by object-oriented frameworks for this purpose [Lew95].

Since a framework provides a pre-defined class hierarchy, object interaction, and thread of control, developers must fit their applications into the framework. This means that in framework-based development, the static structure and dynamic behavior of the framework must first be understood and then adapted to the specific requirements of the application. It is assumed that the effort to learn the framework and develop code within the system is less than the effort required to develop a similar system from scratch.

Although it is recognized that the effort required learning enough about the framework to begin coding is high [Boo94, Pre95, Tal95], little work has been done in the way of minimizing this learning curve.

3.1.1.2 Use Case Maps

The Use Case Maps (UCMs) notation is gaining in popularity and in notoriety. Whether you consider them as causal scenarios, as architectural entities, or as behavior patterns, they can help you to describe and understand emergent behavior of complex and dynamic systems.

The basic idea of UCMs is very simple and is captured by the phrase *causal paths cutting across organizational structures*. The realization of this idea produces a lightweight notation that scales up, while at the same time covering all of the foregoing complexity factors in an integrated and manageable fashion. The notation represents causal paths as sets of wiggly lines that enable a person to visualize scenarios threading through a system without the scenarios actually being specified in any detailed way. Compositions of wiggly lines (which may be called behavior structures) represent large-scale units of emergent behavior cutting across systems, such as network transactions, as first-class architectural entities that are above the level of details and independent of them (because they can be realized in different detailed ways).

The notation is intended to be useful for requirements specification, design, testing, maintenance, adaptation, and evolution. Already, UCMs have been used in a number of areas [Ucm99]:

- Requirements engineering and design of:
    - Real-time systems
    - Object-oriented systems
    - Telecommunication systems
    - Distributed systems
    - Multimedia systems
    - Agent systems
- Detection and avoidance of undesirable feature interactions
- Performance analysis and prediction
- Evaluation of architectural alternatives
- Functional testing

- Detection of race conditions
- Documentation of standards
- Synthesis of message sequence charts and formal specifications
- Reverse-engineering of different systems
- Etc.

Figure 3(d) shows a simple UCM [Ucm99] where a user (Alice) attempts to call another user (Bob) through some network of agents. Each user has an agent responsible for managing subscribed telephony features such as Originating Call Screening (OCS). Alice first sends a connection request (req) to the network through her agent. This request causes the called agent to verify (vrfy) whether the called party is idle or busy (conditions are between square brackets). If he is, then there will be some status update (upd) and a ring signal will be activated on Bob's side (ring). Otherwise, a message stating that Bob is not available will be prepared (mb) and sent back to Alice (msg).

A scenario starts with a triggering event and/or a pre-condition (start point req) and ends with one or more resulting events and/or post-conditions (end points), in our case ring or msg. Intermediate responsibilities (vrfy, upd, mb) have been activated along the way. In this example, the responsibilities are allocated to abstract components (boxes Alice, AgentA, Bob and AgentB), which could be seen as objects, processes, agents, databases, or even roles, actors, or persons.

The construction of a UCM can be done in many ways. For example, one may start by identifying the responsibilities (Figure 3(a)), although not necessarily with diagrams like this one. Responsibilities can then be allocated to scenarios (Figure 3(b)) or to components (Figure 3(c)). Components can be discovered along the way. Eventually, the two views are merged to form a bound map (Figure 3(d)).

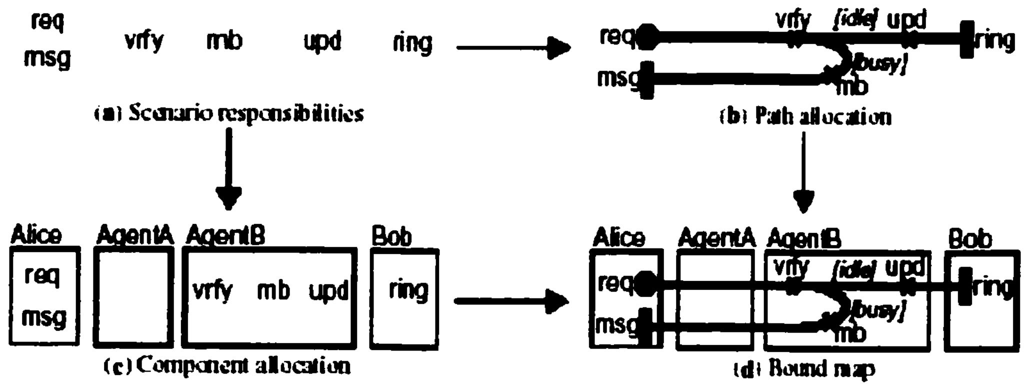


Figure 3. Use Case Maps Construction

Under an apparent simplicity, UCMs such as Figure 3(d) convey a lot of information in a compact form, and they allow for requirements engineers and designers to use two dimensions (structure and behavior) to evaluate architectural alternatives for their system.

3.1.1.3 Use Case

Jacobson introduces use cases in [Jac92] by describing them as: "a behaviorally related sequence of transactions in a dialogue with the system". Perhaps some examples will help to illustrate the notion of use case. Use cases for a word processor might include building an index or inserting a picture. In this way they correspond to menu commands. They can be quite large (building an index) or quite small (making some text bold). Often they might not involve a single command. A use case might be to ensure that the text in a document is consistently formatted; there is no command for this, but this is the use case that drives the need for style sheets.

This latter example introduces many of the difficulties of getting good use cases. The art is to identify the users' goals, not the system functions. One way of doing this is to treat a user's business task as a use case, and to ask how the computer system can support it.

Use case diagrams (Figure 4) provide a way of describing the external view of the system and its interactions with the outside world. In this way it resembles the context diagram of traditional approaches. In this representation the outside world is represented as actors. Actors are roles played by various people, or other computer systems. The emphasis on roles is important: one person may play many roles, and a role may have many people playing it. Use cases are then typical interactions that the actor has with the system.

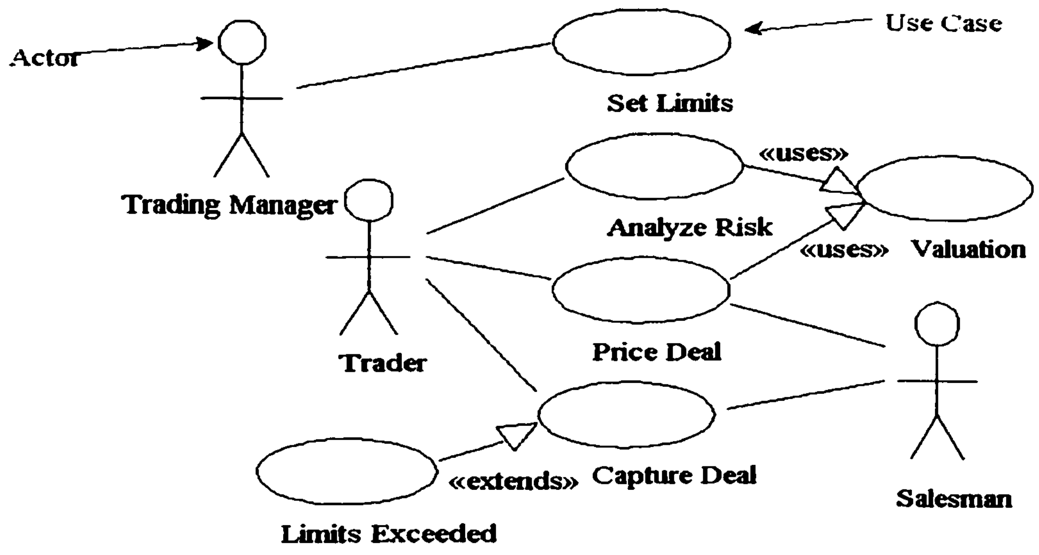


Figure 4. Use case diagrams

Use cases can be seen as a large unstructured set or they can be structured in some way. Jacobson [Jac92] provides two structuring mechanisms. The first allows behavior used by several use cases to be pulled out into a separate use case, which is used by the other ones. This is somewhat like pulling a common subroutine out which is shared by other routines. The second construct allows one use case to extend another. The new use case defines certain points in the original use case at which it takes over with new behavior (it has been likened to a programming patch). Extensions are often used to show exception behavior and special cases, which would otherwise bloat the amount of use cases in the model (Graham [Gra95] uses side-scripts for the same purpose). Use cases act as the structuring mechanism for interaction diagrams. Typically an interaction diagram is drawn for each use case in later design.

One of the big dangers of use cases is that of structuring the software to mimic the use cases. Use cases provide an external view of the system; the software is often structured in a completely different way. The biggest danger is that of turning each use case into a procedural controller, which acts upon simple data holders. When using use cases remember that they are an external view only.

When to Use Them?

Use cases are a vital part of OO development. They should be used when one wants to understand the requirements of a system. Use cases are valuable if just kept on a database as an unstructured list. Each needs a name and a few paragraphs of description. They are central to planning the evolutionary development process. They should also drive system testing and functional testing.

3.1.2 Bottom-up

For the bottom-up visualization techniques, we use an example environment to describe the basic concepts. The MOOSE (Montreal Object-Oriented Slicing Environment) project was developed to provide programmers with support for various cognitive models during the comprehension of large software systems. A cognitive model describes the comprehension processes and knowledge structures used to form a mental representation of the program under examination. Although substantial progress has been made in tool-based environmental support for aiding program comprehension, there is no general structure that classifies different comprehension tools and techniques. [Ril01a]

The MOOSE framework architecture is based on four major components as shown in Figure 5. These components are: (1) Software visualization providing higher level of abstraction; (2) The hybrid slicing framework providing algorithmic support to allow for a reduction of the software complexity; (3) Application framework that combines and utilizes both algorithmic techniques and the various software visualization approaches; and (4) A repository to store and retrieve static and dynamic information. [Ril01b] But for this section bottom-up, they still describe the (1) Software visualization providing higher level of abstraction.

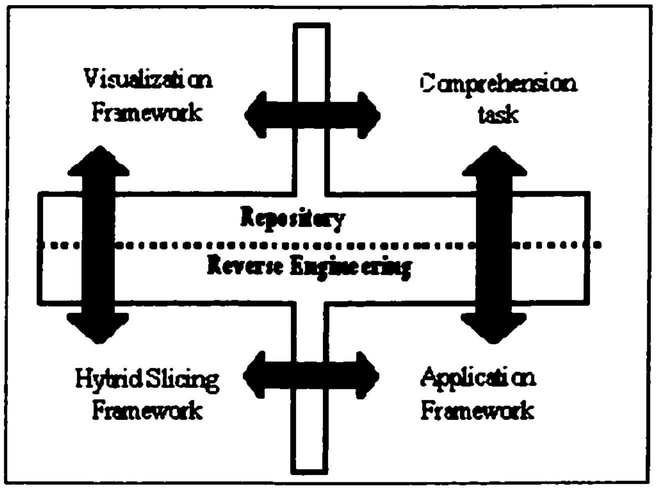


Figure 5. The MOOSE Comprehension Framework

The MOOSE environment provides a descriptive environment that uses reverse engineering to present different levels of visual abstraction to support both bottom-up and top down cognitive comprehension models, as well as static and dynamic visualization techniques. The environment supports two of the better-known static visualization approaches for bottom-up comprehension: the call-graph representation for functional programs and the class-model representation (UML notation) for object-oriented programs. These static abstraction levels are derived by reverse engineering existing source code. Static visualization techniques can help a software engineer to determine the interaction among objects and provide a general understanding of the system structure and the dependencies within the system. However, static analysis does not provide any means for determining how many objects of a class might exist during run-time, or how many method calls might occur between particular objects. In the MOOSE comprehension environment, encoding techniques for dynamic trace information are included that makes it possible to analyze dynamic data at various abstraction levels. The dynamic information is based on information collected as the software system executes. When a comprehension task requires views that involve a larger number of classes, the usability of the dynamic approaches degrades, as it tends to display complex interactions among multiple objects that lead to a very large amount of data to be visualized. To overcome this complexity problem, an architecturally oriented visualization approach was introduced by Steindl [Ste99] that enhances the dynamic visualization techniques by providing both coarse and fine-grained level views. [Ril01a]

The MOOSE environment provides views with different granularity levels and presents new visualization techniques for observing and analyzing the dynamics of program executions. For this environment, the standard UML notion of a sequence diagram and collaboration diagram are reused and extended by applying a reengineering process to derive the *dynamic sequence* (Figure 6) *and dynamic collaboration diagram*. The standard UML sequence and collaboration diagrams are based on use cases for the purpose of determining the later class design. These dynamic diagrams are based on information collected during program execution and they support both cognitive models, the bottom-up as well as the top-down approach to program comprehension. To represent the information across a system execution, a sequence of executed statements is used. Depending on the leve! of visual abstraction, each cell represents an executed statement, function or even class. In the dynamic sequence diagram a program object will be include on the X axle after its first execution. The lifeline is used to represent the timely sequence of program executions. The execution sequence can easily be observed and analyzed based on the position of the cells within their associated lifelines. The visualization techniques support additional enhancements for the comprehension of large programs and long program executions by offering different levels of visual abstractions. Available options include: re-executing a program in a step-wise fashion, stepping through the program execution at the statement/method or class level. During the step-wise re-execution of the program, the execution position in the sequence diagram and the corresponding source code display will be updated dynamically. As part of the UML notation, collaboration diagrams are the primary source of information used to determine class responsibilities and interfaces. MOOSE extends this standard UML notion of a collaboration diagram further to provide an inside view on the dynamics of the relationships and interaction among different classes. The dynamic collaboration diagram has limitations in visualizing the interaction for large programs and their executions. Collaboration diagrams tend to be better suited for depicting simpler interactions among a smaller number of objects and shorter program executions. As the number of objects and messages grows, the diagram becomes increasingly complex and difficult to read.

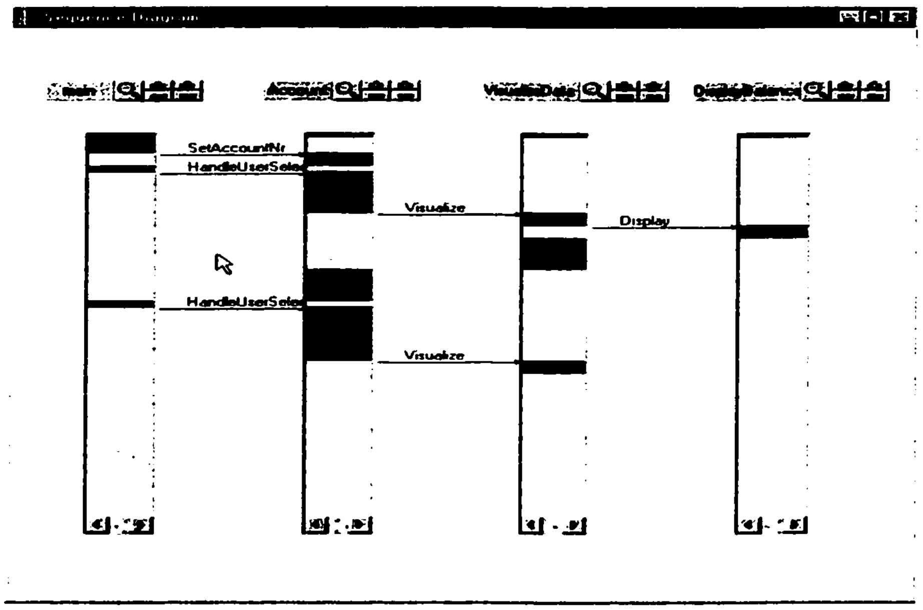


Figure 6. MOOSE Dynamic Sequence

The visualization techniques presented in the MOOSE environment support the cognitive comprehension of software systems and focus on an approach by bottom-up providing the software engineer with the ability to switch from the source code view to the corresponding higher levels of visual abstraction and visa versa. It should be noted that the MOOSE environment is not limited to the presented visualization techniques and that it can be extended easily to support and utilize other visualization techniques like: 3-D, virtual reality, etc. [Sta99]

3.2 Algorithmic Support

Introduction

As described earlier, software visualization techniques are one of the pillars of the cognitive comprehension framework. Software visualization allows for the transformation of a large amount of data to a higher level of abstraction that improves the comprehension of the overall program structure. However, even with higher levels of visual abstractions, a user might still have to deal with a large amount of data without having any meaningful insights about the relationships and the dependencies within a given scenario. Filtering and interpreting enormous quantities of information is a problem for humans. From a mass of data they need to extract knowledge, which will allow them to make informed decisions.

3.2.1 Top down

As we mentioned before, a top-down approach begins with a pre-existing notion of the functionality of the system and proceeds to earmark individual components of the system responsible for a specific task. Similar to the top down algorithms, there are some types of top down algorithm technique, like formal methods, UCMs that can be usually applied in this field.

3.2.1.1 Formal specification

Generally speaking, a *formal specification* is the expression, in some formal language and at some level of abstraction, of a collection of properties some system should satisfy. This purposely general definition covers different notions dependent on what the word "system" really covers, what kind of properties are of interest, what level of abstraction is considered, and what kind of formal language is used.

Formal specifications have been considered for a long time. In the late nineteen forties, Turing observed that reasoning about sequential programs was made simpler by annotating them with properties about program states at specific points [Ran73]. In the late sixties, Floyd, Hoare and Naur proposed axiomatic techniques for proving the consistency between sequential programs and such properties, called specifications [Flo67, Hoa69, Nau69]. Dijkstra showed how a formal calculus over such specifications could be used constructively to derive non-deterministic programs that meet them [Dij75]. Specific techniques were also proposed to formally express intended properties for special kinds of programs, notably, data structured programs [Par72, Lis75] and concurrent programs [Pnu77]. This was the starting point for a whole new area of research aimed at specification-in-the-large [Par77, SRS79, Abr80, Hen80]. The interest in formal specifications and their multiple uses in software engineering have been growing continually since that point [Win90, Cra93, Hin95, Cla96, Win99, SCP2K].

Formal specifications may refer to fairly different things in the software lifecycle; the wording is thus heavily overloaded. An additional source of confusion stems from the fact that a single word is used for a product and the corresponding process.

Complex software applications are built using a series of development steps: (a) high-level goals are identified and refined until a set of requirements on the software and assumptions on the environment can be made precise to satisfy such goals; (b) a software architecture, made of interconnected software components, is designed to satisfy such requirements; and (c) the various components are implemented and integrated so as to satisfy the architectural descriptions. All along this development/satisfaction chain, knowledge about the application domain is often used to guide the elaboration and to support the validation with respect to upstream prescriptions.

The "system" being specified may be a descriptive model of the domain of interest; a prescriptive model of the software and its environment; a prescriptive model of the software alone; a model for the user interface; the software architecture; a model of some process to be followed; etc. The "properties" under consideration may refer to high level goals; functional requirements; non-functional requirements about timing, performance, accuracy, security, etc.; environmental assumptions; services provided by architectural components; protocols of interaction among such components.

Beyond such different realizations of the general concept of specification, there is a common idea of specifications pertaining to the *problem domain* (as opposed to the solution domain). To make sure some solution solves a problem correctly, one must first state that problem correctly. This dichotomy is, however, simplistic; a solution to a problem may in general be given as a set of sub problems to be specified and solved in turn [Swa82]. A specification must thus in general satisfy some higher-level specification and be satisfied by some lower-level specifications.

"Formal" is often confused with "precise" (the former entails the latter but the reverse is of course not true). A specification is *formal* if it is expressed in a language made of three components: rules for determining the grammatical well-formedness of sentences (the syntax); rules for interpreting sentences in a precise, meaningful way within the domain considered (the semantics); and rules for inferring useful information from the specification (the proof theory).

The latter component provides the basis for automated analysis of the specification. The collection of properties being specified is often fairly large; the language should thus allow the specification to be organized into *units* linked through *structuring relationships*- such as specialization, aggregation, instantiation, enrichment, use, etc. Each unit in general has a declaration part, where variables of interest are declared, and an assertion part, where the intended properties on the declared variables are formalized. Formal specification techniques essentially differ from semi-formal ones (such as dataflow diagrams, entity-relationship diagrams or state transition diagrams) in that the latter do not formalize the assertion part.

Formal methods are methods that are valid by virtue of their form, using mathematically well-defined objects and relationships. A formal method provides a means to construct an executable program that can be demonstrated mathematically to be a valid equivalent to its specification. This research theme explores the use of formal methods both in the forward engineering and reverse engineering directions; formal methods may assist with software maintenance but do not require that the system was previously written using such methods, although this is a research thread [Formal 00].

Formal Methods attempt to provide mathematical underpinning for the design of computer systems (hardware or software). A formal method should provide a specification language which has a firm mathematical semantics and a development notion which has a clear concept of what needs to be proved for a design (ultimately implementation) to satisfy its specification. [Bac00]

Examples of specification languages include VDM-SL, Z and RSL from the RAISE project. Development methods include VDM and the RAISE method. Z itself is only a specification language but attempts have been made to support the development process by using various refinement calculi. One could also consider Jean-Raymond Abrial's Abstract Machine Notation as a development method that is related to Z at least by having the same prime originator.

3.2.1.2 UCMs

Use Case Maps are used to describe and integrate use cases representing the requirements. The construction of UCMs can reveal problems with the use cases, which may be incomplete, incorrect, ambiguous, inconsistent, or at different levels of abstraction. UCMs include high-level design information (internal responsibilities and components), but they do not commit to messages between components (in contrast with MSCs), so they are more easily maintainable as design scenarios. UCMs excel at integrating individual features and at the same time allowing for reasoning about potential undesirable interactions. UCMs are not executable as is, but they can be manually translated to a model that allows for fast prototyping and validation. LOTOS is especially well suited for representing UCMs. Mappings to hierarchical finite state machines (used in UML-RT) and to Layered Queuing Networks (for performance modeling) also exist.

UCMs can also serve as a basis for the definition of abstract validation test suites based on the design. This represents a level of completeness different from (and often better than) plain functional testing.

Finally, the use of the UCM Navigator tool enables the automated generation of documentation and of XML code, which can be processed for further analysis and potentially for partial generation of formal models.

3.2.2 Bottom-up

The comprehension of source code plays a prominent role during software maintenance and evolution. There are varieties of support mechanisms for aiding program comprehension, which can be grouped into three categories: unaided browsing, leveraging corporate knowledge with experience, and computer-aided techniques like reverse engineering. In the following sections we focus on the latter and how reverse engineering in combination with algorithmic support can be applied effectively in program comprehension [Agr90, Ril98].

One approach to improve the comprehension of programs is to reduce the amount of data to be observed and inspected. Programmers tend to focus and comprehend selected functions (outputs) and those parts of a program that are directly related to that particular function rather than all possible program functions. One approach is to utilize program slicing, a program decomposition technique that transforms a large program into a smaller one that contains only statements relevant to the computation of a selected function. The notion of program slicing originated in the seminal paper by Weiser [Wei84].

Typically, a program performs a large set of functions/outputs. Rather than trying to comprehend all of a program's functionality, programmers will focus on selected functions (outputs) with the goal of identifying which parts of the program are relevant for that particular function. Program slicing provides support during program comprehension, by capturing the computation of a chosen set of variables/functions at some point (static slicing) in the original program or at a particular execution position (dynamic slicing).

Static slicing

Based on the original definition of Weiser [Wei84] the slice is defined for a slicing criterion $C=(x, V)$, where x is a statement in program P and V is a subset of variables in P. Given C, the slice consists of all statements in P that potentially affect variables in V at position x. Static slices are computed by finding sets of indirectly relevant statements, according to data and control dependencies. The program dependence graph (PDG) was originally defined by Ottenstein and Ottenstein and later refined by Horwitz et al. Data and control dependencies between nodes form a program dependence graph. The static slice of a program with respect to a variable v at a node i, consists of all nodes whose execution could possibly affect the value of the variable v at node i. A static slice can be constructed from the PDG by traversing backwards along the edges of a program dependence graph starting at a node i. The nodes visited during the traversal constitute the program slice.

The major characteristic of static slicing is: the static nature of the source code analysis. This technique is rather inexpensive with respect to run-time overhead and utilization of system resources. Further, it helps in comprehending the overall program dependencies of the selected function/variable at a point of interest. However, static slicing has limitations with respect to the accurate handling of dynamic language constructs (like polymorphism, pointers, aliases, etc.) and conditional statements. In these cases, static slicing algorithms have to make conservative assumptions with respect to these language constructs resulting in larger program slices.

Dynamic slicing

Dynamic program slicing overcomes these shortcomings of static algorithms by utilizing actual program flow information for a particular program execution. This leads to a more accurate handling of dynamic and conditional language constructs and therefore to smaller program slices. As described by Korel in [Kor94], a slicing criterion of program P executed on program input x is a tuple $C = (x, y^q)$ where y^q is a variable at execution position q. A *dynamic slice* of program P on slicing criterion C is any syntactically correct and executable program P' that is obtained from P by deleting zero or more statements. The program P', executed on program input x produces an execution trace T'_x for which there exists the corresponding execution position q' such that the value of y^q in T_x equals the value of $y^{q'}$ in T'_x. In other words, the dynamic slice P' preserves the value of y for a given program input x. Most of the existing dynamic slicing algorithms use data and control dependencies to compute dynamic program slices.

One of the major requirements of dynamic slicing is that it is necessary to identify relevant input conditions for which a dynamic slice should be computed. A commonly used approach to identify such input conditions is referred to as an operational profile, a well-known concept that is frequently applied in testing and software quality assurance.

Removable blocks

Korel introduced in [Kor94] the notion of removable blocks that are described as the smallest component of the program text that can be removed during slice computation without violating the syntactical correctness of the program (e.g. assignment statements, input and output statements, etc.). For the hybrid-slicing framework, they refine the original definition of a removable block, as follows:

"A removable block is a set of user defined statements containing one or more statements included in the scope of each programming language construct, upon removal of the same will not affect the flow of execution" [Kor94].

Each block B has a regular entry to B and a regular exit from B referred to as *r-entry* and *r-exit,* respectively. In unstructured programs, because of jump statements, execution may enter a block directly without going through its r-entry; in this case, we say execution enters the block through a *jump entry.* Similarly, an execution can exit a block without leaving it through its r-exit rather than through its *jump exit.* Intuitively, a block may be removed from a program if its removal does not "disrupt" the flow execution on some input x. Traditional dynamic slicing algorithms (based on data and control dependencies) identify those actions in the execution trace that contribute to the computation of the value of variable y^q. Algorithms based on the notion of removable blocks, on the other hand, identify actions that do not contribute to the computation of y^q. The larger the number of actions that can be identified as "non-contributing", the smaller the computed dynamic slice.

3.3 Application support

Traditionally, in order to understand a program execution, a programmer uses conventional debuggers that support breakpoint facilities and step-wise program execution. Conventional debuggers, however, do not provide any means for identifying those parts of a program and its execution that contribute to the computation of a particular function [Gal91]. Software comprehension is the process of acquiring knowledge about a computer program that supports such activities as bug correction, enhancements, reuse and documentation. Based on the variety of comprehension tasks, different application support should be made available [Kun95]. MOOSE define application support as applications that utilize the underlying repository to achieve a general goal, e.g. design evaluation, functional optimization, etc. Within the application sub-framework it provide an open architecture, based on predefined access interfaces that allow the plug-in of new application support.

3.3.1 Top down

As Scott Tilley described [Til98]: The behavioral approach is top down and inductive, using a goal-driven method of hypothesis postulation and refinement based on expected artifacts derived from knowledge of the application domain. The top-down approach begins with a pre-existing notion of the functionality of the system and proceeds to earmark individual components of the system responsible for specific tasks.

Forward engineering can be considered a top down technique method. Forward engineering is the traditional process of moving from the requirements of the system to its design, and from design to the concrete implementation of the system utilizing domain knowledge. Actually, forward engineering means the same as engineering. The adjective "forward" is only used to distinguish the term from reverse engineering [Har00].

For program comprehension, documentation is often considered a poor source of information. And it's always wrong and doesn't provide the required information. As Scott R. Tilley mentioned [Til96], most of the existing program comprehension tools allow the user to generate documentation that is consistent with the techniques (e.g., data flows, control flows, etc.) available in tools. While the tools can no doubt produce reams of information, it is not clear what additional value (beyond that provided by the tool display) this documentation has, since it can all be reproduced by the tool at any point. In addition, when using the tool on-line (as opposed to documents produced by the tool), the user is assured of consistency with the source code.

In addition to producing hard copy, new program comprehension tools often allow the forging of links between source code and documents [Til96]. An illustrative example [Til96] is the ParaSET workbench, which allows the user to create three types of links between code and documentation: hypertext links which allow forward and backward navigation, ``soft" associations which spawn warnings when documents have become inconsistent with source, and ``hard" associations which force consistency between documents and source for constructs like tokens and identifiers.

ParaSET also supports the forging of links between source and other related information, such as regression tests.

Linking of source and related information can potentially be quite useful for supporting program comprehension, particularly for connecting domain information with source manifestations [Til96]. The value of the support obviously depends on the quality of the information connected, the links built, and the willingness of personnel to build and maintain the links. Ideally, developers would construct the links as the application is built, and maintainers would use and update these links as appropriate. This, however, requires that both developers and maintainers are willing to use the workbench, and that the workbench is appropriate for both tasks.

For the document-centered approaches, research has focused on improved document organization, content and access to information. Also Scott R. Tilley mentioned: [Pinto and Soloway88] focused their work on supporting an 'as needed' strategy toward program comprehension (i.e., learn that portions of the application are necessary to make the expected modification). The authors approach was to design a documentation format that explicitly documented the causal relationships between non-connected sequences of source code. In effect, their documentation format ignores much of the intervening detail and explicitly documents the purpose, structure and effect of the basic algorithm within the program (as opposed to providing this information in a separate document).

3.3.2 Bottom-up

In general, in order to understand a program execution, programmers usually use conventional debuggers to support breakpoint facilities and step-wise program execution. Breakpoints allow a programmer to specify places in a program where the execution should be suspended and to examine various components of a program state. The step-wise execution allows the programmer to observe the program execution for a particular program execution and to comprehend the program flow of the program. Conventional debuggers, however, do not provide any means for identifying those parts of a program and its execution that contribute to the computation of a particular function [Gal91]. Frequently, this leads to the observation of a large amount of unrelated program executions. In this section, third pillar of MOOSE comprehension framework has been presented, which is the application support for program comprehension. Application support addresses issues concerning utilizing the presented algorithmic approaches in connection with presented visualization techniques. Dynamic program slicing not only allows for a reduction of the program slice, but it can provide additional guidance during comprehension of program executions. A dynamic slice was traditionally displayed to the programmer by highlighting the relevant statements in the original program or by removing all statements from the original program that do not belong to the slice. However, for large software systems, this highlighting/reduction might only provide limited guidance to the programmer because the size of the reduction might be still very large. In what follows, will present visualization approaches for program slices, and the presented program slicing related concepts. These applications support a variety of cognitive models and allow the user to switch easily from one model to another [Ril01a].

Class model slice

In the class model slice, a class/member function will be included in the slice if at least one statement within the member function/class is part of the program slice (relevant to the computation of a selected variable or function). The class model slice also includes any cascading relationships among classes that are relevant to the slice (Figure 7 illustrates a class model slice by highlighting relevant statements and classes). Program slices on the class-model slice level are visualized by highlighting these parts of the class model that are included in the program slice.

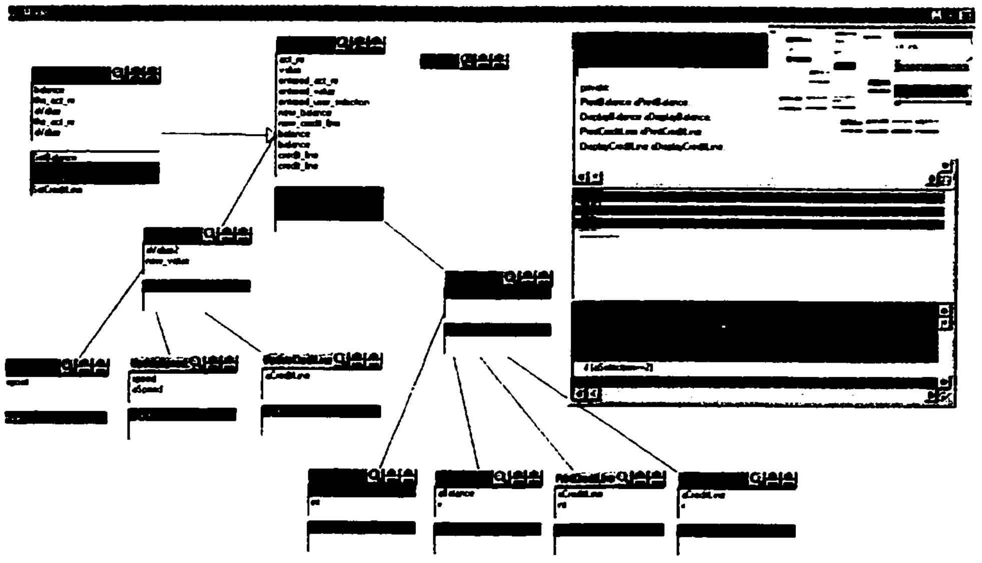


Figure 7. Reverse engineered class model with slice display from MOOSE

Examples of currently implemented application support include static and dynamic class model slice, the dynamic sequence diagram and influencing program artifacts.

Influencing and relevant program artifacts

An important question that arises during the comprehension of software systems for users is to identify which program artifacts should be observed in order to gain a clear understanding of a program's behavior. In [Kor97] the concept of influencing variables was introduced on the source code level and in [Kor98, Ril98] extended for all *influencing program artifacts*. The concept of *influencing program artifacts* allows a user to identify those program artifacts that currently influence a variable/function of interest at a particular program position. For a programmer, it is almost impossible to determine which program artifacts are currently influencing the computation of a particular function by stepping through the source code. Certain parts of a program execution are not relevant to the computation of a particular function and a programmer may only be interested in stepping through these program executions that are relevant to the computation of that particular function. Therefore, MOOSE have developed slicing related concepts that are based on dynamic slicing that allow users to make these distinctions with respect to the function of interest. The concept of *relevant program artifacts* is based on information that is normally discarded after a slice computation. Note that it is possible for a program artifact to be executed multiple times and only some of its executions might be relevant to the computation of the function of interest. A programmer cannot distinguish between relevant and non-relevant executions of a program artifact by observing the source code.

4 MOOSE - A Case Study

Program comprehension is a crucial part of system development and software maintenance. It is expected that a major share of systems development effort goes into modifying and extending preexisting systems, about which we usually know little [Dem99]. The complexity of existing software systems is increasing and their comprehension at the same time is becoming more and more aggravating for those who must maintain these systems.

In this section, a case study, based on the MOOSE (Montreal Object-Oriented Slicing Environment) [Ril01] project is presented. The study presented an example of current research efforts to derive new comprehension tools that take advantage of existing comprehension techniques. In particular, the MOOSE project focuses on the integration of the different top-down and bottom-up comprehension techniques to guide programmers during typical comprehension tasks. The MOOSE project provides an open software comprehension and maintenance framework [Ril01], with its architecture being based on five major components: (1) a task and user-centered approach that will guide users during comprehension of specific tasks, (2) an application framework that provides a set of applications supporting various comprehension tasks, (3) an algorithmic framework, providing analysis and metric functionality, (4) the visualization support, and (5) an underlying repository that provides a communication and interaction interface among all the parts of the environment.

The major motivations for the MOOSE environment are to achieve two goals. The first goal is to provide a suite of tightly integrated tools with a set of coherent functionality. The second goal is to create an open environment that can easily be extended with new tools, algorithms, and applications to meet future demands. The two goals are achieved by creating a general framework that consists of several sub-frameworks as illustrated in Figure 8. The focus of the case study is on the provision of a task and user-centered approach that takes advantage of top down and bottom up comprehension techniques. It introduced a method that aims to increase the functional cohesion of comprehension tools from a user perspective and moves responsibilities away from the user towards the supporting comprehension environments to support an opportunistic approach as applied by end users.

In general, current program comprehension tools provide a suite of tools that aim for "a one tool fits all" comprehension and user needs strategy. However, the user must determine how and when to apply these tools in a coherent and meaningful way. Present software comprehension tools and techniques need to be enhanced to meet current and future demand with respect to their usability and learnability [Rob00, Sef99, Sef01]. The task and user-centered approach, MOOSE project presented a method that aims to increase the functional cohesion of comprehension tools from a user perspective and moves responsibilities away from the user towards the supporting comprehension environments. It also allows combine both bottom up and top down approaches from an end-use perspective.

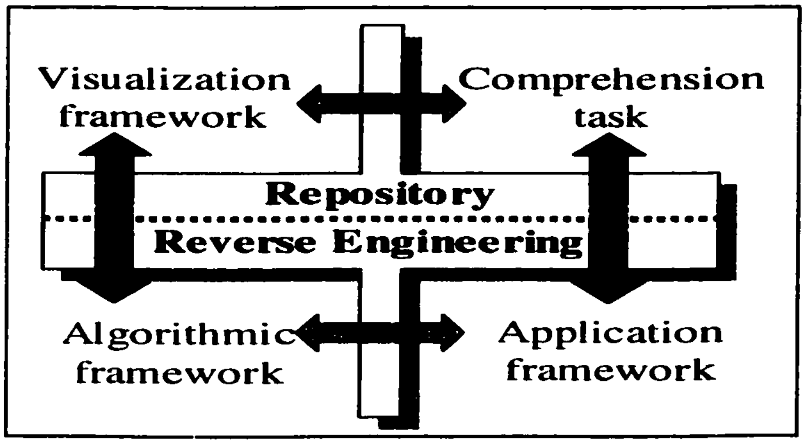


Figure 8. The MOOSE comprehension framework and its components

4.1 Task-centered program comprehension

Many studies have been conducted to observe how programmers understand programs. As a result, several cognitive models have been developed to describe the behavior of these programmers. As part of typical software comprehension processes, users frequently perform a series of related functions that are based on past experience, recurring work patterns, or practices as determined by company policies. Incorporating such recurring work patterns in program comprehension environments will significantly enhance these environments by eliminating repetitive steps and make the comprehension task more transparent and coherent. Novice users are frequently overwhelmed and challenged by identifying the appropriate tools and menu options for a specific task. Within the MOOSE environment, it implemented the following task-centered approaches to enhance the usability and to help to create a more coherent comprehension environment.

Context sensitive menus

A widely used technique in most GUI application is the use of context sensitive menus. They provide a first step towards a task-centered approach by displaying only those functions in the menu bar that are applicable for that particular visualization technique. The context sensitive menus reduce the number of options to choose from and therefore may help to streamline the execution of a particular task.

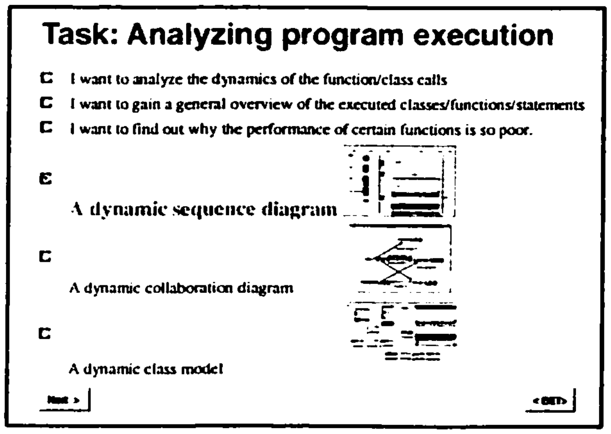
Personalized menus

Personalized menus disclose information in a progressive fashion by streamlining the interaction as skill levels advance and allow the interaction to be customized. Personalized menus show only basic and frequently used commands on short versions of the menus rather than all possible menu options. They also allow for a significant reduction of the user's memory load, in particular with respect to common tasks.

Task wizards

Understanding a software program is often a difficult process because of missing, inconsistent or even too much information. The source code often becomes the sole arbiter of how the system works. Questions are the basis of user's interaction with the documentation (source code) and the system expert comprehension tool. The task to be performed determines the type of maintenance technique/method to be used. Once a user decides on a particular task, the appropriate tools and methods selection has to take place. The main factors that affect software comprehension are the complexity of the problem solved by the program, the program text, the user's mental ability and experience and the task being performed. User tasks can be defined as the operations the user wants to perform on a set of topics.

For example, users may want to identify relevant components, comprehend how the system is currently working, observe relationships among program parts or analyze program executions. Goals are associated with each task. Explicit task support is critical for both novice and expert users, increasing the learn ability and the usability of the system. Software comprehension is the reconstruction of logic, structure and goals that were used in writing a program.



Task: Analyzing program execution

- I want to analyze the dynamics of the function/class calls
- I want to gain a general overview of the executed classes/functions/statements
- I want to find out why the performance of certain functions is so poor.

A dynamic sequence diagram

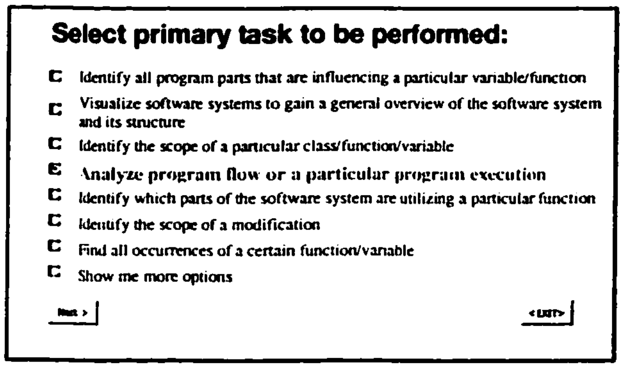
A dynamic collaboration diagram

A dynamic class model

Figure 9. Choosing a comprehension task in MOOSE

A wizard-based approach in connection with the repository provides users with guidance during the switching process among the cognitive models and it fosters the effective application the visualization techniques introduced earlier. This approach also creates a coherent set of functions that are grouped together to achieve a specific goal/task. Previous research [May92] has shown that the process of software comprehension is influenced by several factors, including the knowledge level of the programmer, the size of the program, the user's experience, the complexity of the software, the expected task that is to be performed, and the switching among different cognitive models, techniques and abstraction levels. In our MOOSE prototype, we have incorporated a task taxonomy where each task is associated with a particular comprehension goal (Figure 9).

The available selection is based on information retrieved from the knowledge repository and represents results from previous tasks. However, it should be noted that our MOOSE prototype currently supports only a rather small and narrow subset of software maintenance tasks, by providing wizards that will guide users through these specific program comprehension tasks (Figure 10). In the case that a user task is not supported by the system, the user can cancel the wizard at any time and access all the available visualization, algorithmic and application approaches manually.



Select primary task to be performed:

- Identify all program parts that are influencing a particular variable/function
- Visualize software systems to gain a general overview of the software system and its structure
- Identify the scope of a particular class/function/variable
- Analyze program flow or a particular program execution
- Identify which parts of the software system are utilizing a particular function
- Identify the scope of a modification
- Find all occurrences of a certain function/variable
- Show me more options

Figure 10. Selecting among a coherent set of functions in MOOSE

One advantages of the task-oriented approach is the reduction of the cognitive and memory load. Task wizards can be based on past best practice can enforce company wide standards, and they can provide guidance for the less experienced user.

4.2 A user-centered approach

Evidence culled from day-to-day experience tends to indicate that, in most cases, software engineering technology can meet all the business benefits and requirements and yet still be quite challenging to use and learn. Many definitions of usability exist, often making usability a confusing concept. Generally speaking, usability of software refers to its ease of use and its ease of learning. For an inexperienced user, ease of use is coupled closely with ease of learning and does not necessarily imply a high performance in task completion, whereas experienced users are interested in completing a wide range and number of tasks with minimal obstruction. A usable system would, therefore, be easy to use over time while gaining experience, and easy to learn [Wal98]. Despite efforts made by managers to render the transition more "user-friendly", the associated help documentation and training material, although precise and perfectly describing the product, are often delivered in an esoteric and unreadable language. This could contribute to the rejection of the product by its users. It also explains a large part of the frequently observed phenomenon of modifying the product or the documentation after it has been deployed.

Software, including these tools used by software engineers, is becoming more and more complex, harder to use effectively and more time consuming to learn efficiently. Current training approaches must be aligned with new methods of apprenticeship that are capable of empowering and sustain self-learning and collaborative training. Learning must focus on learner's needs and skills and on how to complete tasks within a specific work context [Sef01].

Furthermore, designing for learnability should not focus only on first-time users (anyone who has never been involved in the process before, such as a newly-hired employee), but also support more experienced users who need to learn advanced tasks as well as new tasks. Occasional users are those who may have been trained, but are involved in the process so infrequently that they may forget how to complete a task between times. Tools should address "on-the-spot" and "just-in-time" assistance for learning and performing tasks. Maintenance tools should support case based problem solving and learning (Figure 11).

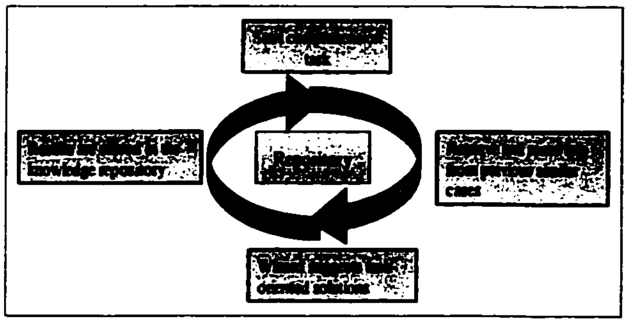


Figure 11. MOOSE and knowledge management

Programmers and software engineers are specializing as part of their daily work routine on recurring tasks during which they can frequently apply case based reasoning techniques. Reasoning techniques include those based on previous cases from within a certain application domain, knowledge from other problem domains, and analogy based reasoning. Maintenance environments should support different types of reasoning.

5 Conclusions

Software comprehension is a crucial phase during system development. It is a well-known fact that a major share of systems development effort goes into the modification and extension of pre-existing systems, about which we usually know little. There are many techniques and methods in this area that can be applied to deal with it. But each of these techniques or methods has their own characteristic and different application areas. Software comprehension is utilizing various techniques and approaches that can be grouped into three main areas: Software visualization techniques to provide higher level of visual abstractions, algorithmic support to provide additional meaning to the information to be displayed and application support to guide programmers during typical program comprehension tasks. There exists a large amount of literature presenting different approaches and techniques applicable for program comprehension, however none of the existing publications provides a structured framework to group existing comprehension techniques. The presented report introduces a novel categorized of comprehension techniques based on mental models from the cognitive sciences. The report presents a survey of state-of the art program comprehension techniques and their grouping into top-down and bottom-up approaches within the visualization, algorithm and application categories.

A case study, based on the MOOSE project is presented to illustrate the current research trends in develop new comprehension tools that take advantage of existing comprehension techniques. The MOOSE project provides users and software developers with guidance during the comprehension of software systems. The framework is based on an open architecture that supports a variety of cognitive models to allow users to take advantage of the various properties of these models. The framework currently supports, but is not limited to the bottom-up, top-down and opportunistic cognitive comprehension models.

There is a number of challenging additional problems that will need to be addressed by future research in the area of software comprehension and tool development. Some of the key areas identified that can be identified after surveying existing comprehension techniques are:

There exists currently only limited tool support for top-down program comprehension. The majority of comprehension tools do not yet take advantage of top-down information like, UCM, Use Cases, and formal specifications, to guide programmers during comprehension tasks. Tools should support program comprehension both bottom-up and top-down at the same time for all categories. In industry, maintainers of large-scale programs frequently switch between these different models depending on the problem-solving task at hand. Current tools are limited to provide knowledge of the implemented source code. There is a clear need to integrate domain knowledge into comprehension tools, to provide maintainers not only with knowledge about the problem domain, but additionally with information about certain business rules and company policies that might be reflected in the source code.

Furthermore, already existing bottom-up comprehension (in particular algorithmic and visualization) techniques have to be further enhanced to be able to cope with large amount of data, distributed environments, and the overall higher complexity of today's and future software systems.

Reference

[Abr80] J.R. Abrial, "The Specification Language Z: Syntax and Semantics". *Programming Research Group*, Oxford Univ., 1980.

[Agr93] H.Agrawal, R.DeMillo, and E.Spafford., "Debugging with dynamic slicing and backtracking", *Software – Practice and Experience*, 23(6), 1993,pp. 589-616

[Bac00] http://www.cs.man.ac.uk/fmethods/background.html

[Bro83] Reven Brooks, "Towards a theory of the comprehension of computer programs", *International Journal of Man-Machine Studies*, 18:543-554, 1983.

[Boe81] Barry W. Boehm. "Software Engineering Economics", *Prentice Hall*, 1981.

[Boo94] G. Booch, , "Designing an application framework", *Dr. Dobb's Journal*, vol. 19, no. 2, February 1994.

[Bas87] V. Basili, R. Selby, "Comparing the effectiveness of software testing strategies", *IEEE Transactions on Software Engineering*, vol. SE-13, no. 12, pp.1278-1296, December 1987.

[Bas96] V. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Soerumgaard, M. Zelkowitz, "The empirical investigation of perspective-based reading"; *Empirical Software Engineering - An International Journal*, vol. 1, no. 2, 1996.

[Bec94] Beck, K.; Johnson, R. "Patterns Generate Architectures". *Procs. ECOOP'94*, Bologna, Italy, Berlin: Spring-Verlag, 1994. p. 89-110.

[Cla98] Claire Knight "Visualization for Program Comprehension: Information and Issues", *Computer Science Technical*, Dept. of Computer Science, University of Durham Report 12/98-October 1998

[Cam96] Marcelo Campo, Claudia Marcos and Alvaro Ortigosa, "Framework Comprehension and Design Patterns: A Reverse Engineering Approach", UNCPBA,1996.

[Cra95] D. Craigen, S. Gerhart and T. Ralston, "Formal Methods Technology Transfer: Impediments and Innovation", in *Applications of Formal Methods*, M.G. Hinchey and J.P. Bowen (eds.), Prentice Hall, 1995, 399-419.

[Cla96] E.M. Clarke, J.M. Wing et al, "Formal Methods: State of the Art and Future Directions", *ACM Computing Surveys Vol. 28 No. 4*, December 1996, 626-643.

[Dem99] Demeyer S., Stéphane Ducasse and Michele Lanza, "A Hybrid Reverse Engineering Platform Combining Metrics and Program Visualization", In Proceedings of WCRE'99, IEEE Computer Society Press, pp.175-187, 1999.

[Dij75] E.W. Dijkstra, "Guarded commands, nondeterminacy and the formal derivation of programs", *Comm. ACM* Vol. 18, August 1975, 453-457.

[For00] http://www.dur.ac.uk/CSM/themes/formal/

[Flo67] R. Floyd, "Assigning Meanings to Programs", In. Mathematical Aspects of Computer Science, *Proc. Symp. Appl. Maths., Vol. 19, American athematical Society*, 1967, 19-32.

[Fje83] R.K.Fjeldstad and W.T.Hamlen. "Application Program Maintenance Study: Report to Our Respondents." Procesdents. *Proceedings GUIDE 48*, Philadelphia, PA, April 1983.

[Gal91] Gallagher, K. and Lyle, J., "Using program slicing in software maintenance", IEEE, *Transactions on Software Engineering*, 17(8), pp. 751-761, 8/1991.

[Gra95] Graham I., "Migrating to Object Technology", *Addison-Wesley*, Wokingham UK, 1995.

[GXL01] http://www.gupro.de/GXL/

[Har00] Marrit Harsu "Re-engineering Legacy Software Through Language Conversion", *Department of Computer and Information Science, University of Tampere*, Tampere 2000, Finland.

[Hen80] K.L. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and their Application", *IEEE Transactions on Software Engineering* Vol. 6 No. 1, January 1980, 2-13.

[Hoa69] C.A.R. Hoare, "An Axiomatic Basis for Computer Programming", *Comm. ACM* Vol. 12 No. 12 No. 10, Oct. 1969, 576-583.

[Hin95] M.G. Hinchey and J.P. Bowen (eds.), *"Applications of Formal Methods"*. Prentice Hall, 1995

[Jac92] Jacobson I., Christerson M., Jonsson P. and Övergaard G. *"Object-Oriented Software Engineering: a use case driven approach"*, Addison-Wesley, 1992.

[Kor97] Korel B. and Rilling, J., "Application of Dynamic Slicing in Program Debugging", *Third International Workshop on Automated Debugging (AADEBUG'97), pp. 59-74,* Linköping, Sweden, May 1997.

[Kor94] B.Korel, and S.Yalamanchili, "Forward Derivation of Dynamic Slices", *Proceedings of the Intern. Symposium on Software Testing and Analysis,* Seattle, 1994, pp. 66-79.

[Kor98] Korel, B. and Rilling, J., "Program Slicing in Understanding of Large Programs", *Proceedings of the 6th International Workshop on Program Comprehension,* IWPC '98, Ischia, Italy, June 1998, pp. 145-152

[Kun95] Kung D., J. Gao et al., "Developing an object-oriented software testing and maintenance environment"; *In Com. of the ACM, Vol. 38, Issue 10* (1995), pp. 75-87.

[Lis75] B.H. Liskov and S.N. Zilles, "Specification Techniques for Data Abstractions", *IEEE Transactions on Software Engineering* Vol. 1. No. 1, March 1975, 7-18.

[May92] Mayrhauser A, Vans M., "An Industrial Experience With an Integrated Code Comprehension Model", *Technical Report CS-92-205),* Ft. Collins, Co, Colorado State University, 1992.

[May94] Mayrhauser A, Vans M, "Program Understanding – A Survey" *Technical Report, CS-94-120,* Colorado State University, Aug. 1994, *pp. 17*

[May98] Mayrhauser A., A. M. Vans, "Program Understanding Behavior During Adaptation of Large Scale Software", *6th IWPC '98,* Italy, 6/1998, *pp. 164-172*

[Mil56] G. A. Miller, *Psychological Review,* 63 (2): 81-96,1956.

[Nau69] P. Naur, "Proofs of algorithms by General Snapshots", *BIT* Vol. 6, 1969, 310-316.

[Par72] D.L.Parnas, "A Technique for Software Module Specification With Examples", *Comm. ACM* Vol. 15, May 1972.

[Pen87] N. Pennington, "Stimulus structures and mental representations in expert comprehension of computer programs". *Cognitive Psychology, 19:295-342,* 1987.

[Pnu77] A. Pnueli, "The Temporal Logics of Programs", *Proc. 18th IEEE Symp. On Foundations of Computer Science*, 1977, 46-57.

[Por95] A. Porter, L. Votta, V. Basili, "Comparing detection methods for software requirements inspections: a replicated experiment", *IEEE Transactions on Software Engineering*, vol. 21, no. 6, 1995, pp.563-575

[Pre95] W. Pree, "*Design Patterns for Object-Oriented Software Development*", Addison-Wesley Publishing Co., 1995.

[Ran73] B. Randell, *"The Origin of Digital Computers"*. Springer-Verlag, 1973.

[Ril01] Juergen Rilling "Maximizing Functional Cohesion of Comprehension Environments by Integrating User and Task Knowledge", *Department of Computer Science, Concordia University,* 2001.

[Ril98] Rilling J, "Investigation Of Dynamic Slicing And Its Application In Program Comprehension", *Ph.D. Thesis; Illinois Institute of Technology,* July 1998.

[Ril01a] Juergen Rilling, "MOOSE-A Cognitive Software Comprehension Framework Based On Reverse Engineering", www.cs.concordia.ca/~feculty/Rilling/, *Department of Computer Science, Concordia University.*

[Ril01b] Juergen Rilling, "A Hybrid Program Slicing Framework", www.cs.concordia.ca/~feculty/Rilling/ Department of Computer Science, Concordia University, 2001

[Rob00] Robitaille R. S., Reinhard Schauer, and Rudolf K. Keller, "Bridging Program Comprehension Tools by Design Navigation". *ICSM'2000 Proceedings*, San Jose, CA, October 2000. IEEE, pp 22-32

[Rug95] Spencer Rugaber, "Draft – 2, Program Comprehension", *Georgia Institute of Technology*, May 4, 1995, *pp. 1*

[Sta99] Staples M. and Bieman, J., "3-D Visualization of Software Structure". In *Advances in Computers*, Volume 49, Edited by M. Zelkowitz, Academic Press, London, 1999.

[SCP2K] "Science of Computer Programming", Special Issue on *Formal Methods in Industry*, Vol. 36 No. 1, January 2000.

[Sef99] Seffah A. "Training Software Developers in Critical Skills", *IEEE Software Magazine*, 6/1999.

[Sef01] A. Seffah and J. Rilling, "Learnability support: Learnability-centered design for software engineering tools", *8th World Conference on Continuing Engineering Education, May 12 - 16,* 2001 Sheraton Centre, Toronto, Canada.

[Shn79] B.Shneiderman and R.Mayer. "Syntactic/semantic interactions in programmer behavior: A model and experimental result". *International Journal of Computer and Information Science,* 1979, *8(3):219-238*

[Sim00] S. E. Sim, R. C. Holt, R. Koschke, "WoSEF Workshop on Standard Exchange Formats", *ICSE 2000 Workshop proceedings,* Limerick 2000 (http://www.cs. toronto.edu/~simsuz/wosef/).

[Sol84] E.Soloway and K. Ehrlich, "Empirical studies of programming knowledge". *IEEE Transactions on Software Engineering,* Sept. 1984, *SE-10 (5): 595-609.*

[SRS79] "Proceedings SRS - Specification of Reliable Software", *IEEE Catalog No. 79 CH1401-9C,* 1979.

[Sta99] Staples M. and Bieman, J., "3-D Visualization of Software Structure". *In Advances in Computers, Volume 49,* Edited by M. Zelkowitz, Academic Press, London, 1999.

[Ste96] Seifka M., A Sane and R.H. Campbell, "Architecture-oriented visualization". *In Proceedings of the 1996 ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'96),* 1996, pp.389-405.

[Ste99] Steindl C., "Static Analysis of Object-Oriented Programs", *9th ECOOP Workshop for PhD Students in OO-Programmiong,* Lisabon, Portugal, June 14, 1999.

[Sto98] Margaret Anne D. Storey, "A Cognitive Framework for Describing Evaluating Software Exploration Tool", Technical Report, *Simon Fraser University,* 1998.

[Tal95] Taligent Inc., "The Power of Frameworks", *Addison-Wesley,* 1995.

[Til96a] Scott R. Tilley, Santanu Paul, "Toward a Framework for Program Understanding", *Software Engineering Institute, Carnegie Mellon University, Center for Software Engineering, IBM T.J. Watson Research Center,* 1996.

[Til96] Scott R. Tilley, "Perspectives on Legacy System Reengineering", Carnegie Mellon University, 1996, http://www.sei.cmu.edu/reengineering/pubs/lsysree/node155.html

[Til98] Scott Tilley, "A reverse engineering Environment Framework", Technical Report, *CMU/SEI-98-TR-005, ESC-TR-98-005*, April 1998, *pp. 5.*

[Ucm99] http://www.usecasemaps.org/

[Wal98] Walker, R.J., Murphy, G.C., Freeman-Benson, B., Wright, D., Swanson, D., Isaak, J., "Visualizing Dynamic Software System Information through High-level Models", *Proceedings of OOPSLA'98Vancouver, October 1998. Published as SIGPLAN Notices 33(10)*, October 1998, *pp. 271-283.*

[Win90] J.M. Wing, "A Specifier's Introduction to Formal Methods", *IEEE Computer* Vol. 23 No. 9, September 1990.

[Win99] J.M. Wing, J. Woodcock and J. Davies (eds.), "FM-99 -World Conference on Formal Methods in the Development of Computing Systems", *LNCS 1708 and 1709*, Springer-Verlag, 1999.

[Wei84] M.Weiser, "Program slicing", *IEEE Transactions on Software Engineering* 10(4), 1984, pp. 352-357.