Verbal Source Code Descriptor

By

Sima Soudian Tehrani
B.Sc., Shahid Beheshti University, 1996

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE
in
THE FACULTY OF GRADUATE STUDIES
(Department of Electrical and Computer Engineering)

We accept this thesis as conforming
to the required standard

The University of British Columbia
January 2003
© Sima Soudian Tehrani, 2003
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Electrical and Computer Engineering

The University of British Columbia
Vancouver, Canada

Date January 8, 2003
ABSTRACT

Program comprehension and understanding is a demanding cognitive task in software maintenance. Program understanding involves the construction of a mental model of the software.

Researchers introduced various cognitive models for program understanding, such as the top-down model [8], bottom-up model [7], the knowledge-based model [9], the systematic and as-needed program understanding model [11], and the integrated program understanding meta-model [12]. Many tools have been developed to assist program comprehension based on aspects of the above models. These tools basically create visual high-level information about the software under investigation, particularly at the structure and design level, and are designed to form the mental models of the software in hand.

This research presents a new approach to program comprehension with the goal of reducing the cost of existing tools and overcoming some of their limitations, such as language dependency, and the insufficiency of providing information for complete program understanding. Program understanding information basically exists at the time of programming, and can be captured with less cost and effort; however some problems, such as mental loading problem, prevent programmers from providing such information in the form of written documentation. I have developed a verbal documentation strategy to reduce the overhead of typing at the time of programming. The hypothesis is that we can capture more useful understanding information from the programmer using this strategy.

The Verbal Source Code Descriptor (VSCD) System is introduced as a tool to help programmers document their source code verbally, while writing the code. A preliminary evaluation with 12 programmers was carried out to evaluate the feasibility and ease of using the VSCD System. This is compared with more conventional implementation documenting techniques, which is still in the form of putting comments inside the source code.

The results indicate the VSCD system may be useful and efficient in generating comments. Future studies must verify the quality of VSCD comments and investigate the efficiencies of using this strategy to make program comprehension more flexible and faster. These studies may combine the VSCD strategy with existing methods to investigate the effects of this approach.
Table of Contents

ABSTRACT ........................................................................................................................................ ii
LIST OF TABLES ................................................................................................................................ v
LIST OF FIGURES ........................................................................................................................ vi
ACKNOWLEDGMENTS .................................................................................................................... vii
PREFACE ........................................................................................................................................... viii
INTRODUCTION ............................................................................................................................. 1

CHAPTER 1: PROBLEM DEFINITION ............................................................................................. 3
1.1. MOTIVATION AND GOALS ............................................................................................... 4
1.2. GENERAL APPROACH .................................................................................................... 5
1.2.1. Project Location and Setting .................................................................................. 5
1.2.2. Methods .................................................................................................................. 5

CHAPTER 2: LITRATURE SURVEY ............................................................................................. 6
2.1. PROGRAM COMPREHENSION ......................................................................................... 6
2.1.1. Introduction ............................................................................................................ 6
2.1.2. Definition ............................................................................................................... 7
2.1.3. Models of Human Comprehension of Software .................................................. 7
2.1.4. Program understanding classification ................................................................ 11
2.2. PROGRAM COMPREHENSION TOOLS AND TECHNIQUES ........................................ 13
2.2.1. The effectiveness of program understanding tools .......................................... 13
2.2.2. Preventive Techniques and Tools ..................................................................... 14
2.2.2.1. Literate Programming .................................................................................. 14
2.2.2.2. Elucidative Programming .......................................................................... 14
2.2.2.3. Aspect-Oriented Programming .................................................................. 17
2.2.3. Posterior Techniques and Tools ...................................................................... 18
2.2.3.1. Program Analysis Techniques-Basics and Approaches ............................. 18
2.2.3.1.1. Static Techniques .................................................................................... 18
2.2.3.1.1.1. Textual Analysis ................................................................................. 18
2.2.3.1.1.2. Lexical Analysis ................................................................................ 18
2.2.3.1.1.3. Syntactic Analysis ............................................................................. 19
2.2.3.1.1.4. Control Flow Analysis .................................................................... 19
2.2.3.1.1.5. Data Flow Analysis ......................................................................... 19
2.2.3.1.1.6. Program Dependence Graph (PDG) .............................................. 19
2.2.3.1.1.7. Slicing ................................................................................................. 20
2.2.3.1.1.8. Cliché Recognition ......................................................................... 20
2.2.3.1.1.9. Abstract Interpretation ................................................................. 20
2.2.3.1.2. Dynamic Analysis .............................................................................. 20
2.2.3.1.3. Partial Evaluation ............................................................................... 20
2.2.3.1.4. Static and Dynamic Feature Analysis .............................................. 21
2.2.3.2. Program understanding using reverse engineering ..................................... 21
2.3. PROGRAM READABILITY ............................................................................................... 23
2.3.1. Examples of programming styles and their influences on enhancing readability 23
2.3.1.1. Commenting .............................................................................................. 23
2.3.1.2. Variable Names ......................................................................................... 23
2.3.1.3. Indentation ................................................................................................. 24
2.3.1.4. Code Colorizing ....................................................................................... 24
2.3.1.5. Other techniques ....................................................................................... 24
List of Tables

Table 3.1: System Configuration.................................................................31
Table 3.2: Forms Descriptions.................................................................32
Table 3.3: The structure of LinkInfo table in the database ......................33
Table 3.4: The structure of FilePaths table in the database .....................33
Table 4.1: Type of comments.................................................................48
Table 4.2: Number of comments per comment sub-category for control and VSCD groups........48
Table 4.3: Ratings of importance by comment sub-category .....................49
Table 4.4: Comment’s quality rating scales definition ..............................51
Table 4.5: Mean and standard deviation for quality categories .................52
Table 4.6: Samples of source code descriptions generated by VSCD participants........53
Table 4.7: Mean and SD for mental workload sub-categories ratings and overall workload ....54
List of Figures

Figure 2.1: Program Understanding Classification ................................................. 11
Figure 2.2: The layout of panes in an elucidator ....................................................... 15
Figure 2.3: A control flow graph (a) and its control dependence sub-graph (b) [60] .... 19
Figure 3.1: The VSCD system architecture .............................................................. 31
Figure 3.2: One to many relationships between table FilePaths and LinkInfo .......... 34
Figure 3.3: VSCD Main Screen ................................................................................. 37
Figure 3.4: The process of using VSCD ................................................................. 42
Figure 4.1: Mean for mental workload sub-scales and overall workload (WWL) .... 54
Acknowledgments

I would like to express my gratitude to all those who gave me the possibility of completing this thesis. I am deeply indebted to Dr. Deborah Fels from The Ryerson Polytechnic University whose assistance, supervision and encouragement helped me in all the times of researching for and the writing of this thesis.

I want to thank Dr. Mabo Robert Ito and Dr. Sidney Fels from the University of British Columbia for their stimulating support.

My colleagues in the CLT lab at Ryerson Polytechnic University supported me in my research work and contributed to my experimental studies. I want to thank them for all their help, support, interest and valuable hints.

Especially, I would like to give my special thanks to my husband Farhad whose patient love enabled me to complete this work.
Preface

This report documents the analysis, design, implementation and experimental results of a prototype that implements a software development environment with the focus on documentation by voice technique. This report introduces the concepts and techniques of program comprehension. Some of the problems in existing program understanding techniques and approaches are considered in the prototype.

The purpose of this report is to test the idea of documentation by voice and feature-based classification of the source code in such a degree to present an overview of their potential. This should serve as a base for the choice of focus in future research work.

The prototype is developed in the software development environment Visual Basic 6.0, made by Microsoft, and employs the IBM voice recognition technology, including IBM ViaVoice release 9.0. Documents are created by the use of Microsoft Word.

In this report, literature references are represented as numbers in square brackets, these numbers can be found in the reference list at the end.
INTRODUCTION

Managing the complexity of software programs is an important issue in software engineering because costs are directly related to the time and effort required to maintain complex software. Complexity increases during the software lifecycle because over time, more functionality is often added and, specifications change, resulting in relevant modifications, and development tool obsolescence forces developers to update code for use with newer versions of those tools [1].

Enhancing and changing old systems makes them less maintainable due to outdated documentation, accompanied by more complexity. Software maintainers usually do not have good supporting documentation, and the source code is often the only resource for gathering information about a software system. An important research activity is thus to investigate and develop possible solutions for simplifying the task of program understanding and comprehension to support the task of software maintenance.

Various techniques, generally categorized into two groups (the “Prevenient” and the “Posterior” approaches [4]), are designed to address the issues of program understanding. The prevenient approach documents the source code with the programmer’s understanding information along with the development of the program. This mostly takes the form of comments, specifications and design documents, and source code descriptions.

Posterior approaches gather and extract understanding information from the existing source code after it is developed. Reverse engineering techniques, including Textual Analysis [6], Lexical Analysis [6], Syntactic Analysis [6], Control Flow Analysis [6, 81], Data Flow Analysis [6, 81], Program Dependence Graph [6, 60], Slicing [6, 61], Cliche Recognition [6], Abstract Interpretation [6, 13], Dynamic Analysis [6, 62], Partial Evaluation [6, 63], and Static and Dynamic Feature Analysis [59] are some examples of such posterior techniques. Posterior techniques can be difficult to apply, are time consuming, and are costly. However, representing understanding information at the time of programming potentially takes relatively less effort. There is little research and few tools available that enable programmers to easily and efficiently create program understanding documentation. One of the most recent approaches is Elucidative Programming, which is a documentation paradigm for maintaining the understanding of software programs. This method provides an environment for programmers to write the source code and documentation in separate files and link them together by hyperlinks. It motivates programmers to write documents in a formatted structure along with programming. However, the drawback of this technique is that organizing and typing comments at the time of programming seems intrusive. Therefore, the extra overhead of such techniques prevented their broad usage in real world programming practices.

My research thus focuses on developing a new “Prevenient” approach, called verbal documentation, for using voice as an alternative input method for
documentation. This technique is inspired by Elucidative Programming and resembles some of its features, especially in the user interface layer.

The Verbal Source Code Descriptor (VSCD) is a tool designed to assist programmers verbally document software. The VSCD is a programming environment comprised of tools and menus for creating a project, writing the source code, verbalizing comments and descriptions, editing and formatting the program and comments, and finally linking the source code to its descriptions. The system is only a prototype for demonstrating the idea of verbal commenting at the time of programming, and its features are extendable for more functionality.

The VSCD works on the premise that as programmers develop code, they can verbalize their thoughts to provide additional information about what they are doing. This information includes the rationale behind programming thoughts and ideas, and also includes some aspects of a complex program, as discussed in Section 2.2.2.3. This premise is similar to the talk-aloud evaluation method commonly used for usability studies [67]. In usability studies, participants are asked to talk aloud their thoughts and say what they are doing, why they are doing it, and what they think the result will be. The talk aloud protocol is believed to provide reliable and accurate data on the positive and negative aspects of hardware and software interface designs (as initial concepts to advanced prototypes) [68].

This thesis provides the background, description of the VSCD development process, and the results of the pilot study carried out to evaluate the usage of the VSCD, and shows how to formally evaluate the system’s usage for future work. Chapter 1 provides the scope and motivation for my research. Chapter 2 gives a summary of the literature of program understanding, and addresses the techniques and issues that have been studied in this area. Chapter 3 describes VSCD system and how it can be used to support entering data and linking processes. Chapter 4 presents the experimental results of a pilot study carried out to provide preliminary data of the system used by programmers, and finally, Chapter 5 provides a discussion and conclusion of the work, and gives an overview of future research.
Chapter 1: Problem Definition

At the time of programming, much detailed information exists about a program that are not being documented and formulated or not being updated. This information is lost in later phases of development life cycle if it is not captured during the programming process. Posterior techniques recapture this lost and missing information from the source code in later phases of the program life cycle, however this can be difficult to apply, and is time consuming and costly. One way to address these significant issues is to carry out more research on techniques which gather “understanding information” from programmers at the time of programming, referred to as “Prevenient” techniques [4].

Normark [4] suggests that four important deterrents prevent programmers from writing the detailed documents at the time of programming:

1. The program readability decreases when long explanations and comments are used between lines of source code.
2. There must be an accurate and reliable mechanism to define the relationship between the program and its documentation if it is kept in separate files.
3. Programming tasks require high cognitive effort that can completely occupy a programmer’s attention. Documentation tasks that are expected to be carried out concurrent with programming tasks require additional cognitive and attentive resources that may not be available from the programmer. As a result, programmers often ignore documentation tasks in favour of programming tasks.
4. Programmers might not have enough motivation to do documentation in the absence of immediate and obvious payback.

In an attempt to overcome some of these deterrents, several techniques and methods, such as “Elucidative Programming” [4] and “Literate Programming” [3] are introduced. Capturing the essential understanding from programmers and relating those understandings to relevant program units are major goals of these techniques. These approaches suggest new techniques for writing documents rather than traditional ways of inserting comments into source code. Details of these two techniques are explained further in Chapter 2.

However, the mental workload of writing a program is considerably high, and adding the extra overhead of writing and structuring documentation on top of this often leads to the programmer to refrain from documentation.

The Verbal Source Code Descriptor (VSCD) extends the concepts of Elucidative Programming by specifically addressing the mental workload problem, and by using programmer’s voice as a more natural input method than typing. It also uses feature-based classification technique, as discussed in Sections 2.2.2.3 and 2.2.3.1.4. Based on these ideas, following questions arise:
1. To which degree is it possible to motivate programmers to perform documentation verbally?

I need to find out how natural and easy documentation is by voice, while writing the source code, and how programmers would see this technique in their future development environments.

2. How can the tool be designed to facilitate the task of documentation by voice?

3. Whether or not the task of programming and documentation can be done simultaneously?

4. How much the primary evaluation of this technique in terms of quality of comments and workload would be promising for future work?

The pilot study provides preliminary data that can be used to make assumptions to guide future work.

The following questions are not addressed in this thesis, but remain interesting however, if we intend to design in the future. This technique tends to help document programs for future programmers. It is important to verify the assumption of creating high quality documents by this technique with actual software maintainers. Future experiments should consider the following questions as a guide to their work.

1. How easy would it be to understand a program that is being written in verbal documentation framework?

2. Is the understanding information obtained using this technique sufficient for software maintenance, or does it have to be combined with other techniques?

3. To which degree is it possible to expand this tool to support a multi-user paradigm?

4. How would the system scale to a complex programming task with many modules?

5. What are the impacts of feature-oriented classification technique on programming and maintenance?

1.1. Motivation and goals

The goal of this research is to study the information that can be captured from programmers at the time of writing the source code. This includes both the information put into the source code as regular comments, and some additional information that never gets documented, such as the programmer's thought flow and process. The task of commenting and documentation in its traditional form is time consuming for programmers, and a voice recognition editor makes it easier to write comments by voice than typing in standard environments. The similar technique is used in “programming by voice” to prevent “Repetitive Stress Injury”, such as “Carpal Tunnel Syndrome” [70]. To achieve this goal, I designed
and implemented a system that generates an environment that enables people to program and document by voice with affordable overhead, and then conducted a pilot study as a guide for future studies. A method of determining if the system is successful is also proposed.

1.2. General Approach

1.2.1. Project Location and Setting

The location for implementing this project was in the CLT lab of Ryerson University in Toronto. The project is implemented using a computer PC, IBM Pentium II and RAM 64 MB. The system's platform is Windows 98, and the development environment is Microsoft Visual Basic 6.0. The voice recognition technology used is IBM ViaVoice release 9.0 and Microsoft SAPI 4.0.

1.2.2. Methods

After designing and implementing the VSCD System, an experimental study was designed and developed to evaluate whether programmers could successfully use the system, and the impact of the technique on the quality and quantity of their comments.

A comparative study was developed to investigate the documentation process using the VSCD System and standard documentation practices. Participants were divided into a control group and a VSCD group, and were asked to do a similar programming task. Participants in experimental group used the VSCD System, and control group participants used the traditional way of programming and commenting the source code in a text editor. Subjective assessments of the ease of use and efficiency of commenting techniques were collected through pre and post-study questionnaires. Mental workload assessments were carried out using the NASA TLX technique. To examine the effectiveness of using the VSCD to produce comments, the quantitative and qualitative analyses of comments generated by participants in each group was also carried out. Detailed results of this study are reported in Chapter 4.
Chapter 2: LITERATURE SURVEY

2.1. Program Comprehension

2.1.1. Introduction

As software develops and matures, it often becomes more complex and convoluted. Managing this complexity is an important issue in software engineering. Complexity is added during a program's life cycle because of the following reasons:

- More functionality is required.
- The software specification changes as time passes and the system must be modified according to new or additional specifications.
- New versions of development tools replaced the old ones, and the source code does not work in new environments. [2]

One important result of enhancing and changing old systems is that these become less and less maintainable due to outdated documentation, accompanied by more and more complexity. Corbi [55] defined "Old" code as existing code that cannot be easily understood. The process of working on old code is defined as software renewal, software evolution, program redevelopment, software renovation, unprogramming, reverse engineering and software maintenance.

Difficulties in understanding and maintaining "old" code arise from lack of familiarity by the current programming team with the system, due to the loss of the original programmer's knowledge, and because of poor and constrained design unrelated to the code's age. Software maintainers usually do not have good supporting documentation, and the source code is often the only resource for gathering information about a software system.

As a result of these difficulties, software maintenance can be the most expensive part of the software life cycle [36]. An important research activity is thus, to investigate and develop possible solutions in order to simplify the task of program understanding and comprehension so that the task of software maintenance can be supported.

Today, software maintenance is often allocated more effort and resources than just building new software [6]. Program understanding and comprehension is one of the major tasks of software maintenance. It is estimated that 50-90% of maintenance time is devoted to program comprehension [36]. Ranbally [23] reports three main aspects of program comprehension that emphasize the importance of this task:

1. It is necessary to fully understand a program to select appropriate test data and interpret the output produced with such data.
2. Comprehension is essential to debug the logical and semantic aspects of a program.

3. Successful modification of a program requires a thorough understanding of the program.

In this chapter I summarized the available techniques and issues in the literature of program comprehension to provide concrete background information for future work. This information can be used to combine existing techniques with the suggested strategy in this research work to evaluate the efficiency of this technique. After providing a formal definition of program comprehension, various program comprehension models and strategies based on the study of software psychology are described. A classification system where program-understanding methods are divided into 2 main categories is discussed, along with how various approaches fit within this classification.

2.1.2. Definition

Program comprehension is “The process of acquiring knowledge about a computer program” [6, pg. 1]. To accomplish this task, programmers commonly use existing documents, such as requirement analysis and design documentation, and review the source code to understand the features of a program. However, the lack of complete and consistent documentation, and the complexity of the source code make program comprehension tasks and activities potentially very difficult and resource intensive. It is also important to recognize that human systems have limitations, which must be considered when developing support systems for program comprehension. Software engineers attempt to resolve these problems by introducing program interpretation strategies and models that are implemented as automated program understanding and software visualization tools and techniques.

2.1.3. Models of Human Comprehension of Software

Shneiderman [7][6, pg. 4] defined the concept of software psychology as “The study of human performance in using computer and information systems”. In order to define the features of a program understanding tool, the human skills and capabilities, and also limitations in program comprehension must be understood. Various program understanding models and methods have been designed to match the various human mental models [5, 6]. However, there is insufficient evidence to demonstrate which approach is most appropriate. There can be large differences in programmer and “understander” abilities that depend on the individual’s level of experience or familiarity with the code, and also the type of task. Also, the understander’s mental model changes as the process of investigating a piece of code progresses [55].

Researchers [5, 6] suggest that there are six main strategies for program comprehension based on human processing models:

- **Bottom-up or Shneiderman model.**
Shneiderman defines program comprehension as "Converting the code of a given program to some internal semantic form" [6, pg. 5]. The Shneiderman model is defined as "Reading the source code and then mentally chunking the low-level software artifacts into meaningful, higher-level abstractions." [7][5, pg. 2]. The task of reading and chunking is divided into 3 levels: [6]

1. Low-level understanding of source code line by line.
2. Mid-level understanding of the functionality of each algorithm and module.
3. High-level understanding of the overall program function.

Experienced programmers have semantic and syntactic knowledge of programming in long-term memory. Semantic knowledge is knowing the purpose of different programming concepts, such as sorting, searching and working with the stack regardless of any specific programming language. Syntactic knowledge means knowing the details of different programming languages, such as loop structure, and variable assignments. A programmer uses both types of information for each of the three levels of program comprehension. The programmer's semantic and syntactic knowledge helps to achieve the conversion of code into some internal model or semantic form.

• Top-down or Brooks' Model

This is defined as "Reconstructing knowledge about the application domain and mapping that to the source code."[8][5, pg.2]. The Brooks' model defines strategies for the comprehension of completed programs, and are based on the following components:

1. Information about the program to be understood, such as source code and related documentation.
2. The knowledge base, as defined by Letovsky [5, 9] (See the knowledge-based approach in this section).
3. A mental model that represents the programmer's current understanding of the program.
4. An assimilation process that updates the mental model while processing the state of understanding.

Brooks' model is based on 3 main assumptions:

1. Programming is the process of mapping between a task domain and the programming domain using intermediate domains.
2. Program understanding processes involve rebuilding all or part of those mappings.
3. The rebuilding process consists of “Creation”, “Confirmation”, and “Refinement of Hypotheses”, which are the descriptions of various domains and their relationships to each other.

Brooks describes the process of comprehension as having the following sub-processes:

An understander constructs a primary hypothesis of what a program does based on existing information she gathered about the program. For example, the program name itself could form part of the primary hypothesis. As the understander begins to carry out program understanding tasks, the primary hypothesis changes and subsidiary hypotheses can be generated. The understander follows this route according to his motivation for program understanding. For example, if the understander intends to fix a bug in a part of the system, he only follows the path(s) to the specific point in the program and ignores other paths. The cascading and producing of hypotheses continues until the understander has sufficient information to map her set of tasks (task domain) to the program domain. At this point the understander can verify the hypothesis against the program code. The visible details in the source code that are used for hypothesis verification are called “beacons” [6]. Beacons can include code artifacts. For example, a typical beacon for the hypothesis “sort” is a pair of loops.

Brooks’ model attempts to account for the origins and causes of the following program understanding observations and experiences:

1. How the functionality of the program being understood affects its comprehensibility.

   The more difficult programs to understand have more complex intermediate hypotheses so there are more intermediate domains between the task domain and the program domain. Sometimes these intermediate domains have little, if any, supporting documents.

2. How is the ease of comprehension affected by the characteristics of the programming language?

   The text and features of each programming language affects comprehension because some programming languages’ feature help to find “beacons” easier and faster. For example, in a language such as C, the hypothesis that the constant variable PI contains 3.1415 is easy to confirm because C has the “const” feature that helps to define constant variables. However, in a programming language without constant declarations, it is hard to confirm that PI is 3.1415 even if it is assigned at the beginning of the program because an understander must confirm PI never changed.

3. How do program alteration tasks, such as debugging or enhancement tasks, affect program comprehension?
The ease of comprehension can be significantly affected by program alteration tasks because the understander only looks for what is most relevant to understanding the program for these specific tasks. For example, a debugger who finds an error in an output task ignores any hypotheses that deal with input or computation operations. A more comprehensive alteration task such as debugging may require a more extensive understanding of the entire program.

4. What is the relationship between the programmer’s skills and abilities and his ability to comprehend a program, and why do some understanders find a program easier to understand than others do?

The understander’s skills and capabilities have an impact on the comprehension task because experiences of the task domain help to construct hypotheses easier and faster. In addition, knowledge of programming language helps to find beacons easier.

- **Soloway’s Model:** [6,10]
  Soloway’s model describes the need to recover the original intention behind the code. Intentions, or goals, and the techniques to realize these intentions, or plans, form the basis of Soloway’s model. Program comprehension is the process of “Finding plans in code, combining these plans to form sub-goals and combining sub-goals into higher level goals.” [6, pg.8]. The programming language semantic, plan knowledge, efficiency knowledge, problem domain knowledge and discourse rules are the components of the knowledge base of this model.

- **The knowledge-based approach** proposed by Letovsky [9] gathers information and clues during top-down or bottom-up processes. Letovsky believes that “Programmers are opportunistic processors capable of exploiting either bottom-up or top-down clues.” [5, pg. 2]. This method is also based on three components as described in the top-down model:
  1. The programmer’s application and programming expertise.
  2. Mental model.
  3. Assimilation process as described in the Brooks’ model.

- **The Systematic and as-needed** model is proposed by Littman et al [11]. He suggests that the understander reads all codes in detail (“Systematic approach”) or only focuses on one part of the code “as needed”. In the Systematic approach, a maintainer first examines the entire program and tries to find the relationships between different modules of the program before starting any modification. This strategy is efficient with small size programs. The “as-needed” strategy is used with larger applications where an understander studies a limited, but relevant to a specific modification, portion of an entire system.
• **Integrated approaches** proposed by Von Mayerhauser and Vans [12] consists of a model that combines top-down, bottom-up and knowledge based models.

There are apparent similarities between all of these program comprehension models and the way that programmers originally plan and design their program. For example, bottom-up and top-down models are similar to design models for structured programming patterns. Assuming that original programmers and maintainers may have a similar mental model of a program's form, and the structure of a program's facts and components, we can expect that the programmers' descriptions of their thoughts to be of interest to maintainers. These comprehension models describe the ways that maintainers construct and extract the original programmers' understanding information. Therefore, if this information becomes available at maintenance time, there can be a considerable decrease in maintainers' mental workload, also decreasing time and cost of maintenance. In my research, I proposed voice as a new source of input for programmers to provide as much understanding information as possible.

### 2.1.4. Program understanding classification

Program understanding approaches can be divided into at least two major categories: the “Prevenient” and the “Posterior” approaches. The prevenient approach is to document the source code with the programmer’s understanding information along with the development of the program. Posterior approaches deal with gathering and extracting understanding information from the existing source code. This classification, along with several illustrative examples, is shown in Figure 2.1.

![Figure 2.1: Program Understanding Classification](image)

At the time of programming much of the detailed information about a program is not being documented and formulated or not being updated, and disappears or is
invalidated in later phases of the development life cycle. As a consequence, reverse engineering techniques were developed to extract the information directly from the source code. The posterior techniques however, are difficult, time consuming, and expensive, while writing down and representing understanding information at the time of programming takes relatively less effort. However, Normark [4] outlines several significant issues that prevent programmers from writing detailed documents at the time of programming:

- **The program comment problem**
  Long explanations in the form of comments between source code lines decrease program readability.

- **The program-documentation relation problem**
  If the documentation is kept in a separate file there must be a definition for the strong relationship between the parts of the source code and their corresponding explanations.

- **The mental loading problem**
  The programming task is a highly demanding cognitive activity that basically requires the programmer’s complete attention. If the documentation process requires additional mental work, the programmer ignores this task.

- **The programmer’s motivation problem**
  “Most documentation efforts are long term investments in relation to program maintenance, and have little immediate payback for the programmer.” [4, pg. 2]

New programming development environments and techniques, such as “literate programming” and “Elucidative Programming” have been developed in an attempt to address these issues. The basic premise of these new environments is to capture the essential understanding from the programmers and relate these to the relevant program units during program development.
2.2. Program Comprehension tools and techniques

In Section 2.1.2 program comprehension is defined as “the process of acquiring knowledge about a computer program”. The importance of software maintenance, and that today companies spend much effort in reusing old software instead of creating new programs, are also discussed as major motivations for the extensive effort in developing program understanding tools.

In this section, the effectiveness of program understanding tools and whether they help programmers understand a program, are presented. The features of various existing commercial and academic program-understanding products are also presented in Appendix F. One reason for explaining all the techniques and methods of program comprehension is to be able to elaborate the strategy of documentation by voice in a more effective way for future work. Later in this section it is suggested that the best program comprehension tool is the one that combines a variety of techniques, and provides the user with enough flexibility to switch between the methods and gather as much information as possible.

2.2.1. The effectiveness of program understanding tools

Often, programmers use various methodologies and tools to understand a program, but a question arises that to what extend a program understanding tool can help programmers to understand a program. The most important aspect of these methods and tools is that a tool must match the mental and cognitive model of doing a task, otherwise it could be frustrating and useless for users. Story et al [5] reported this issue from the results of an experimental study on 30 subjects using 3 different program understanding tools (Rigi, ShriMP, and SNiFF+). Their observations showed that the best tools are those that combine different comprehension strategies that can be switched easily and reduce cognitive overhead as the program is explored. They should include a broad variety of understanding methods without imposing one theory of investigation on the programmer. For example, an effective source code-searching tool, along with representing the high-level views of the source code among the low-level representation of the program with effective hyper links, could help to design a useful program-understanding tool.

There are some important factors that would affect the user’s choice of comprehension strategy. The most important factors are the program to be understood, characteristics of the task to be resolved, and programming expertise and domain knowledge of the software system. These factors should be considered to choose the right program understanding tool.

The programmer must have the choice to select the most appropriate function or process based on his or her expertise and the current assigned task. Tools should let programmers learn how to develop strategies using various kind of basic program information. While remaining flexible, they should make the process of understanding and obtaining information as easy as possible. They should not require complex query commands that could divert the programmer from his or her primary purpose.

This section provides the major features and specifications of some program comprehension techniques and tools in both preventient and posterior categories. This information can be used as a guide for further studies to combine various techniques.
Verbal Source Code Descriptor

with the suggested strategy in this research work in order to provide the most effective program comprehension method.

2.2.2. Prevenient Techniques and Tools

This section presents the design and specifications of some existing software tools that are designed based on prevenient approaches.

2.2.2.1. Literate Programming

Literate programming was introduced by Knuth in 1984 [3]. He believed that the construction of a program should not be considered as a set of instructions to tell the computer what to do, but should first explain to human beings what we want a computer to do [3]. This approach considers a program and its documentation as literature, which are read by programmers in the same way as technical papers. Knuth also implemented “WEB” as a system tool for literate programming that consists of “Weave” and “Tangle” tools. Weave produces printable documentation, and Tangle extracts and composes the program fragments from documentation.

The main contribution of this approach is that program fragments annotate the documentation instead of explanations that usually annotate the source code in the form of comments. The organization of the program focuses on the documentation structure instead of the program structure. This organization requires a linguistic framework for documentation, and a mechanism to assemble the program pieces into a whole program (using some form of interconnection language). This approach has a number of drawbacks that make it impractical in current programming practices.

First, the “mental load” of using this approach is high due to the use of three languages in one document (“Programming language”, “Documentation language”, and “Interconnection language”).

Second, from the programmer’s point of view, the source code is fragmented within the documentation and it is different from the standard compiler presentation. This can cause difficulties when programmers debug the code for syntax or run-time errors.

Third, this approach creates a beautiful documentation suitable for paper representation. However, today’s media and hypertext-oriented representations are more attractive and practical. Also, the WEB system does not consider procedural abstractions, objects, and their relations. As a result, Normark [37] introduced the “Elucidative Programming” approach as a variant of literate programming that uses concepts derived from hypermedia.

2.2.2.2. Elucidative Programming

Elucidative Programming attempts to improve internal program documentation (“The understanding of a program at code level” [71, pg.1]) in order to reduce “costly detective work on the source code using posterior approaches” [4, pg. 3]. The specification of the Elucidative Programming approach involves the following:

1) “The internal documentation must be oriented toward current and future developers of the program” [4, pg. 3].
The main goal of internal documentation is to provide current and future program developers with sufficient and useful understanding information.

2) "The internal documentation must address explanations that maintain the program understanding and clarify the thoughts behind the program" [4, pg. 3].

3) "The program source file must be intact, without embedded or surrounding documentation" [4, pg. 4].

This requirement emphasizes that the organization of the source code and the program structure must be retained intact while practicing Elucidative Programming. This would resolve the problem of assembling the fragmented program to prepare it for compilation, identified in the literate programming approach.

4) "The programmer must experience support of the program explanation task in the program editing tool" [4, pg. 4].

An Elucidative Programming environment requires an editor that allows the source code and its documentation to be expressed separately, and then linked. It also must provide the user with easy to use navigational tools to accomplish these tasks. The main goal is to minimize the mental load of a programmer when trying to create internal program documentation while generating code.

5) "The program chunking structure follows the main abstractions supported by the programming language." [4, pg. 4]

Each programming language has its own abstraction structure, such as classes, modules, functions and procedures. It is important to keep this abstraction structure while practicing Elucidative Programming, as compared with literate programming where a program chunk is a piece of code that the programmer chooses to explain.

6) "The documented program must be available in an attractive on-line representation suitable for exposition in an Internet browser." [4, pg. 5]

This requirement suggests that the program and documentation can be represented in a tagged HTML format browsable on the Internet or intranets.

Normark [37] explained the concepts and model of an Elucidative Programming environment for the Scheme language in the LISP family of programming languages. This environment consists of a user interface that includes two windows, a Documentation pane and a Program pane, to present the program and the documented program understanding as hypertext side by side (see Figure 2.2).

Menu and index pane

<table>
<thead>
<tr>
<th>Documentation Pane</th>
<th>Program Pane</th>
</tr>
</thead>
</table>

Figure 2.2: The layout of panes in an elucidator
Verbal Source Code Descriptor

In this environment, the program and documentation consists of entities where program source code entities are the building blocks of the programming language (classes, modules, procedures, etc.), and documentation entities are sections and subsections of the program explanation. To refer to a program entity from a document entity and vice versa, the model uses a naming scheme. For example, to refer to a function called "Multiple" located in the file named "File" we can use the name File$Multiple.

Program and documentation are connected by means of "relations" defined as the following:

- **Strong Document-Program relation**: A link between a program entity and its explanation in a document entity.
- **Weak Document-Program relation**: A link between a program entity and a document entity where the program entity is mentioned but not explained.
- **Program-Program relation**: A link between two program entities that represents a name and its definition.
- **Document-Document relation**: A link between two document entities that rely on each other.

There are two important tools that support the creation of entities and their relations in an Elucidative Programming environment; the "Elucidator" tool that presents the program and documentation in HTML format, and "Editor" that provides programmers with facilities to produce a program and its related documentation.

Elucidative Programming has been implemented as the Scheme and Java Elucidator. The Scheme Elucidator is currently being used locally at Aalborg University in Denmark [69]. The usefulness of this technique is also evaluated in student projects and industry [72,73]. Some of the questions that are addressed in the evaluation of this method that can also address some of my research questions, are listed below. The Elucidative Programming can be referred as a method to provide code level understanding information at the time of programming. The VSCD technique is closely related to Elucidative Programming and I would expect a similar experience in some aspects.

1. "Is Elucidative Programming a suitable mean for presenting and evaluating software during reviews?" [72, pg.4]

   This question focuses on the way that Elucidative Programming structures source code and documentation side by side on screen, and allows navigation from source code to documentation and vice versa. The hypothesis is that the program developer can benefit from reviews of the documentation, along with source code reviews. The evaluation of Elucidative Programming showed that students structured the documentation very carefully from the beginning, and described rationales behind the source code in their documentation. However, they found that it is difficult to persuade students to use Elucidative Programming consequently because structuring documents between programming is a time consuming and interruptive task.

2. "Do programs become less error prone when the developer is forced to write Elucidative programs?" [72, pg.4]
Evaluations did not give any concrete answers to this question. However, it is assumed that writing about a program before actually writing the code likely improves the quality of the program because the original programmer is forced to think more about the correctness of the selected solution when trying to explain it.

3. “Does Elucidative Programming ease maintenance?” [72, pg.5]

There is no evidence to evaluate the usefulness of this technique in software maintenance. However, they expect that the documentation in the specific structure can ease maintenance.

As a consequence, the evaluation of Elucidative Programming showed that this technique would encourage programmers to provide more detailed information about the solutions that they choose in their programs, and give them confidence in the correctness of their solutions. However, this seems to increase programmer’s overhead due to writing documentation in an Elucidative structure at the time of programming.

2.2.2.3. Aspect-Oriented Programming

Programming languages and techniques evolved over past decades from machine codes from the earliest computer systems through concepts, such as structured programming and object oriented techniques. Each evolution enhanced the ability to achieve the clear and comprehensible separation of concerns at the source code level. Object-oriented techniques and the idea of abstracting behavior and data types in a single entity called object, has recently been the most dominant programming paradigm [74]. However, many programming problems have been found that cannot be supported either by an object-oriented framework or procedural approach, and these techniques are unable to capture all the design decisions that have to be implemented [75]. In some systems there are requirements called concerns or aspects that cannot be neatly decomposed into common existing frameworks. Aspects tend not to be units of the system’s functional decomposition, but rather properties that affect the performance or semantics of the components in systematic ways. Examples of these concerns include memory allocation, minimizing network traffic, and so on. It is impossible to clearly design and decompose these aspects in OOP or a procedural framework and a programmer should manually tangle the code to reflect these concerns; this makes the system very hard to understand and maintain. The difficulties in localizing concerns in a tangled code arises from the fact that such aspects crosscut the system’s basic functionality instead of being localized within a specific structural piece.

Researchers introduced new techniques and mechanisms called Post-Object Programming (POP), and examples of such techniques include domain-specific languages, metaprogramming, feature-oriented development, views/viewpoints, asynchronous message brokering, and Aspect-Oriented Programming (AOP) [75, 76].

AOP is a new software decomposition method for improving the separation of concerns and simplifying the realization of crosscutting concerns. It also provides some mechanism for weaving aspects and the base code into a coherent system [75].

One of the hypotheses in this research is that documentation by voice provides a mean to capture crosscutting aspects and concerns, along with other information. I also suggested a feature-based classification technique as explained in Chapter 3, to
demonstrate the functional concerns (features) at the source code level. This technique provides a means to classify a set of related software components (classes, modules and functions) that tend to perform a specific task together, and give them a name called feature. Features give semantic meaning to a group of related components to simplify their understanding. It is important that software components be designed in a way to be decomposed based on features. Techniques such as Feature-Oriented Domain Analysis (FODA) [77] that focus on the identification of distinctive features of a software system in a domain can be used to achieve this goal. The feature-based classification technique that is suggested and implemented in VSCD however only supports the one-dimensional separation of aspects and provides a one-to-one relationship between features and program components. Therefore, the current implementation can be used for the programs that only pose functional types of concerns, such as business rules, and does not support crosscutting and non-functional aspects. Further design and implementation is required to address the above issues properly.

2.2.3. Posterior Techniques and Tools

This section demonstrates the features and specifications of some posterior techniques, such as program analysis techniques, and software and program visualization strategies. A summary of the features of some commercial software tools that are designed based on these techniques is also presented in Appendix F.

2.2.3.1. Program Analysis Techniques-Basics and Approaches

Program analysis techniques are software programs that are used to comprehend code after it is generated. Program analysis techniques can be divided into 2 categories:

1. Static (Reading and analyzing the source code).
2. Dynamic (Studying a program while is executing).

2.2.3.1.1. Static Techniques

Static analysis techniques work directly with the source code. They are performed on the source code to analyze and present information about the code and the relationships between various aspects of that code (e.g., variables and values, patterns of variable usage, etc.).

2.2.3.1.1.1. Textual Analysis

This type of analysis measures the comprehensibility of a program in terms of its size. The size could be defined as the number of lines of code, the complexity of control flows, pattern of variable usage, or some other measurement criteria.

2.2.3.1.1.2. Lexical Analysis

This type of analysis is similar to what compilers do in the token recognition analysis phase of compilation. A table of symbols or tokens, such as variables with a description of those tokens' properties, such as name, initial value and length is extracted. For example, lexical analysis can be used to find the average length of variable names.
2.2.3.1.3. Syntactic Analysis

This analysis parses the source code syntactically and allows understanders to explore statements, expressions and modules of a program. The output of the parser is a parse tree, but it contains redundant information, such as punctuation, that is not relevant to program understanding. To address this problem, analyzers eliminate these irrelevant details and construct a second tree called an abstract syntax tree (commonly used by automatic program analyzer tools).

2.2.3.1.4. Control Flow Analysis

This analysis has two forms: “Intra-procedural Analysis” to determine the order in which the statements are executed in a sub-routine, and “Inter-Procedural Analysis” that specifies the calling relationship between program modules.

2.2.3.1.5. Data Flow Analysis

Data flow analysis shows how data definition flows are used in a program. It specifies which statements could be affected by a given assignment statement. Users can find code that never executes, variables that are not defined, or statements that could change after bug fixes.

2.2.3.1.6. Program Dependence Graph (PDG)

This is a refinement of Data Flow Analysis and was introduced by researchers interested in running programs on parallel machines [60]. With this analysis technique, control and data flow dependencies are treated uniformly in a similar representation. “The PDG represents a program as a graph in which the nodes are statements and predicate expressions (or operators and operands), and the edges incident to a node represent both the data values on which the node’s operations depend and the control conditions on which the execution of these operations depend.” [60]. The structural properties of the PDG make it appropriate for program comprehension.

![Figure 2.3: A control flow graph (a) and its control dependence sub-graph (b) [60].](image-url)

For example, the control flow diagram represents the control of the basic blocks of a program, indicated as numbers in Figure 2.3 (a). In these graphs T and F represent True and False, respectively. The corresponding control dependence sub-graph is...
shown in Figure 2.3 (b). This graph shows the regions of control dependence as Rx, and shows where a region depends on the condition and what condition lead to it. In many applications, regions are conceptually more useful than basic blocks.

2.2.3.1.7. Slicing

This method is another derivation of Data Flow Analysis that was introduced by Weiser [61]. A slice of a program for a specific variable at a particular line of code is a part of the program that gives value to the variable [61]. If a variable value were wrong in any line of the code then searching the appropriate slice would be much easier than searching the entire source code.

2.2.3.1.8. Cliché Recognition

This is a form of static program analysis that recognizes similar programming patterns in source code. These patterns are called clichés (or idioms). For example, a pattern describing loops for a linear search is a cliché. Detecting a cliché is difficult due to the different ways of performing linear searches.

2.2.3.1.9. Abstract Interpretation

One final static analysis approach, denotational semantics [13], provides a mathematical technique to specify the semantic properties of a program. In this approach, various data types called semantic domains describe each part of the program. For example, the bindings between variables and values are given by a domain. The meaning of statements of the program is given by a function that specifies what happens in the corresponding domains when the initial domain executes. “For example, the meaning of an assignment statement is a function that maps from the state before the assignment was executed to the state afterwards.” [6, pg. 14].

2.2.3.1.2. Dynamic Analysis

Dynamic approaches [62] help to capture understanding information while a program is executing. This approach is useful for understanding the performance and correctness of a program when they are called, profiled and tested. Profiling determines the number of times that a statement or procedure is executed. Profiling analysis can be done by inserting extra code to do counting, or to interrupt the program execution periodically to monitor what the program is doing and then construct a statistical model [6]. Testing techniques are another form of dynamic analysis. These techniques test a program to make sure every statement is executed.

2.2.3.1.3. Partial Evaluation

Partial evaluation [63] is used to address the problem of understanding large and complicated programs, such as real-time applications used in telecommunications. In this approach, a software tool called “Partial Evaluator” takes a program and its parameters’ values as input, and reduces the size of the program by replacing program statements with the values computed by them. In the case of real-time applications where input parameters are the state of the switch hardware, this approach replaced the system states with global variables. This helps to reduce the complexity of the system and to determine the flow of the program, instead of running the application to simulate execution, which is not applicable in most real-time applications.
2.2.3.1.4. Static and Dynamic Feature Analysis

One of the problems of program understanding is trying to explore how a certain feature of the program is implemented in the source code. In a large application containing hundreds of modules, it can be difficult to locate a component that implements a specific feature. Eisenbarth et al. [59] focused on a feature-oriented approach for finding only the components of interest that are related to a given feature. This technique combines static and dynamic analysis to localize appropriate components related to a set of features. In the dynamic analysis, the program is executed against a specific scenario and reveals the subprograms executed when a specific feature is invoked. In addition to localizing subprograms, a concept analysis is carried out to explore detailed relationships between features and subprograms. Information on this relationship identifies subprograms that are required by a subset of features and classifies these subprograms according to the features it supports. A static analysis follows the dynamic analysis. The purpose of the static analysis is to use the concept analysis information to narrow the dependencies of subprograms and extract ones containing understandable feature-specific components. During this phase of analysis, the utility subprograms that do not contain any application related logic information are identified and sorted out.

However, the understanding information obtained from this feature analysis technique can be easily captured at the time of programming from the original programmer. This is one of the issues that I considered in the design of the VSCD System that is explained in Chapter 3.

2.2.3.2. Program understanding using reverse engineering

Reengineering is defined differently by people. Arnold [50] believes that “Software reengineering is any activity that: (1) improves one’s understanding of software, or (2) prepares and improves the software itself, usually for increased maintainability, reusability, or evolvability.”

GUIDE [51] defines reengineering as: “The process of modifying the internal mechanisms of a system or program or the data structures of a system or program without changing its functionality.”

Chikofsky and Cross [52] defined reengineering as “The examination and alteration of a subject system to reconstitute it in a new form and subsequent implementation of that form.”

The process of reverse engineering does not change the software system, but captures more information about it. The process of reengineering however, consists of reverse engineering followed by a forward engineering phase that starts from the high level abstractions generated at the reverse engineering phase, and moves to the low level physical implementation to alter the program.

One way of facilitating program understanding is through computer-aided reverse engineering [38]. This process involves two distinct phases:

- Identifying the system’s current components and capturing their dependencies.
- Discovering design information and generating system abstractions.

The maintainer must understand the program elements and their relationship. For example, functions, subroutines and procedures are some major elements of a program that are being written in a linear programming environment; however, their
relationship is multi-dimensional (A procedure can be related to other procedures in different ways). Therefore, the tools that allow the user to view these multi-dimensional relationships can simplify the task of program understanding.

Researchers are interested in developing reverse engineering tools for program understanding. The main concerns of these researches are to represent program behavior in a formal way. Reverse engineering has many supporting aspects [38]: “It may focus on features such as control flows, global variables, data structures and resource exchanges. At a higher semantic level it may focus on behavioral features, such as memory usage, un-initialized variables, value ranges, and algorithmic plans. At a higher level of abstraction it may focus on business rules, policies and responsibilities.”

Techniques and methods of reverse engineering must have enough flexibility and scalability. By flexibility, it means the ability to adapt to a variety of situations. The user must be able to extend the tool’s functionality. Scalability means that the approaches of a reverse engineering tool must be applicable to a multi-million-line program.

Rigi is a reverse engineering environment produced by Muller [53] that consists of the following components:

- Parsing Subsystem
- A distributed, multi-user repository
- An interactive graph editor
- A documentation strategy using up-to-date views
- A facility to understand document structure
- An extension mechanism via a scripting language

As mentioned in previous sections, human cognition plays an important role in program understanding. There is always a trade-off between what should be automated and what is to be left to humans. The best is to combine these two. In the Rigi system the extraction phase is an automatic process that uses the parsing subsystem to extract the software system’s artifacts and stores them in the repository. In complex and large systems this phase is followed by a semi automatic process of using human pattern recognition skills.

Rigi exposes a subsystem composition technique to generate layered hierarchies of subsystems. This simplifies the understanding of complex, large systems. The clustering criteria depends on several factors like purpose, audience, and domain.

Another feature of Rigi is re-documentation of the software system using interactive views. It is important to keep the program understanding process cost effective. The created views can lessen the time required to understand the system. As mentioned above, one of the important specifications of a reverse engineering tool is flexibility. The task of reverse engineering is domain dependent (application, implementation and reverse engineering domain), so a good approach must allow users to customize the environment. Rigi provides this capability through a scripting language that let users customize, combine and automate reverse engineering activities.

PUNS (Program UNderstanding Support) is another software reengineering tool developed by Cleveland [54]. This system consists of two distributed components:
Verbal Source Code Descriptor

1. The repository and all associate routines to manipulate the information in the repository.
2. A friendly user interface to present the information obtained from the repository.

PUNS provides users with multiple views of the program; however, there is a problem with this multi-view scheme. It is not easy to move between views in this environment, and the user is forced to use a query language to obtain information about the program, instead of using a point and click interface.

2.3. Program Readability

While analysis techniques can provide a user with some high level means for understanding code, a large factor in interpreting the information resulting from analysis techniques is program readability. Researchers [23] suggest that program readability is defined by the quality of various factors such as variable names, internal documentation, and modularity. In this section, these factors are discussed, along with their influence on program readability and comprehension. A taxonomy of programming styles is presented in Appendix G.

2.3.1. Examples of programming styles and their influences on enhancing readability

2.3.1.1. Commenting

The influence of comments on program understanding is not completely defined, nor well understood. There are different ideas about program commenting in the literature. Studies on short programs by Okimoto [19] indicate that comments are often not useful to programmers and maintainers, especially if they are out of date. Comments, particularly outdated ones, can cause misunderstanding and error in the semantic representation of the code. Shneiderman [20] showed that high-level and functionally descriptive comments, such as the comments describing functions and subroutines in the header of the module, facilitate program understanding. However, low-level comments, such as the comments describing lines of code inside the body of a module, might interfere with code readability and disrupt the flow of the program. This problem is addressed in the Elucidative Programming technique and VSCD by separating the source code from documentation, and defining a mechanism to make appropriate relationships between them. This provides another advantage in that the documentation is not tied to just one specific piece of source code, and therefore, it makes it possible to create one to many relationships between document and source code entities.

2.3.1.2. Variable Names

Using meaningful mnemonic variable names can help to ease the memory burden on the programmer by providing simple ways to remember the meaning and assignments of variables [64].
2.3.1.3. Indentation

The impact of indentation on program understanding has not been well understood. Some researchers, such as Weissman [21] and Love [22] showed that indentation does not improve program understanding, especially in large programs where indentation might cause line-breaks at inappropriate points in the code, thus making the code harder to read.

2.3.1.4. Code Colorizing

Rambally [23] investigated the influence of color on program readability and comprehensibility. He introduced three different color schemes for a program comprehension task in an experiment involving novice and expert programmers. Color scheme A uses different colors to show the scope of loops and conditional statements. For example, one color may represent a control structure in one block and a second color could represent a nested block. In Color scheme B, each program structure is represented by a specific color, such as blue for variable definitions, green for function calls, and purple for I/O statements. The third color scheme is the usual black and white programs. Seventy-nine students divided into two groups of novices and experts were asked to read a Pascal program for thirty minutes in three different color schemes, and rate the difficulty encountered in comprehending the program through a comprehension quiz. Experimental results indicated that the degree of comprehensibility is higher when using color scheme B in which different color codes are used to identify the various statements in a program. The second highest level of program comprehension is achieved when different color codes are used for each block in the program. Black and white programs are the most difficult cases to comprehend. This experiment supports the effectiveness of color for improving program readability and comprehensibility. In the design of VSCD I did not implement this feature; however, it is recommended to consider this feature for the enhancement of this system in future.

2.3.1.5. Other techniques

Some researchers consider the use of flowcharts as a method for understanding the logical structure of a program. A flowchart visually presents a program, its code, documentation and behavior using computer graphics. Baecker [26] proposed seven graphic design principles to enhance program visualization:

1. Paper and page characteristics.
3. Typographic vocabulary (applying appropriate type styles to encode and distinguish various kinds of program tokens.)
4. Typesetting parameters (Text point size, headline size, word spacing, paragraph indentation and line spacing.)
5. Symbolic and diagrammatic elements.
6. Color, texture and value.
7. Meta text (Augment the source text with supplementary text that has been automatically derived from the original source text.)
Verbal Source Code Descriptor

He introduced a software tool called the SEE processor that uses 2 phases to construct a C program flow chart. Phase 1 produces syntax trees for all C constructs, and recognizes comments among the source code. Phase 2 provides output instructions to a tool called TROFF, which is a device independent document formatter, designed by Kernighan in 1982 [56]. The output of TROFF is formatted source code based on the design principles outlined by Baecker [26].

Baecker [26] carried out a series of experiments to evaluate the efficiency and effectiveness of SEE in helping programmers read and understand code. The experiments showed that enhanced source text presentation increased the program’s readability by 25%. Baecker also suggested that there may be alternative methods of program visualization such as “Programs as publications”, “Program animation”, and embedding program’s source in a hypertext system incorporating dynamics, animation, color and sound. Bigelow and Riley [57] also suggested the use of hypertext technology to represent and show the relationships of various aspects of source code. Clifton [25] used connector lines to connect the beginnings to the endings of control structures as one.

Since the early Baecker and Kernighan work on visualization, there has been considerable research carried out on these other visualization techniques under the broader research topic of software visualization [2].
2.4. The bottom line

Program understanding techniques and tools studied in this chapter provide programmers with information for facilitating program comprehension. Most of these techniques, however, tend to recapture “understanding information” missing in design or implementation documents from the source code. The difficulty and cost of using these techniques inspired researchers to focus on providing new development environments for structured, implementation-consistent and comprehensible program documentation. A few software tools have been introduced based on the latest ideas, such as Literate programming and Elucidative Programming; however, none of these became commercial. Too much overhead limits documenting programs this way, and programmers prefer traditional way of inserting comments in the source code once in a while, if they even bother.

I envision a programming environment that not only follows the idea of structured and documentation-oriented programming, but addresses the overhead problem, while introducing documentation by voice as an alternative input method for decreasing overhead of documentation while programming. This will decrease the burden of structuring documents, and allow programmers to concentrate on the task of programming. However, the ability to do both a programming task and verbal documentation at once depends on how much cognitive processing these two tasks require.

Although the task of writing the source code and verbal documentation seem to be two separate tasks that can be done simultaneously, a study of cognitive psychology determines that human cognitive ability to do more than one task at a time is very limited [82].

Some people assume that the human brain is a “Massively parallel” system that can process more than one task at a time [82]. This assumption is not correct. Parallel systems in information technology refer to systems that consist of more than one processor that work independently. However, this definition cannot be applied to the human brain. Based on neurobiological studies “the main thinking part of the brain is the cerebral cortex, and in this there are no separate units at all. Every region in the cortex has direct connections (i.e. send axons and receive axons) to all its neighbors, and in many cases to further regions. Since these connections are direct, each operation of every region in the cortex is dependent directly on the activity of other regions in the cortex, that is, it is not independent.” [82]

However, some evidences in cognitive psychology show that some tasks are capable of overriding interference from other activities. This depends on the amount of attention and cognitive processing that each task requires. Some tasks cannot be done simultaneously, such as listening to two messages at the same time, or processing two visual images at the same time. Some tasks that initially require significant cognitive processing have been shown to become almost automatic, and therefore, do not require as much attention and focus, for example,
typing, CPR, reading, and driving. As these skills are mastered they require less cognitive processing, which allows one to do other things at the same time [83]. The task of programming however, is not only typing, but involves a lot of thinking about solutions and their implementation, so it cannot become automatic, like a simple typing process. Verbal documentation, on the other hand, requires attention for concentrating on explanations and describing solutions. Thus, it is assumed that the process of programming and verbal documentation is a serial task. Chapter 4 presents some evidence from the results of observations during the experimental study. The details of these results are discussed later in Chapter 4.

Despite the results of investigating how simultaneous the task of programming and documentation by voice are, it is expected to decrease the workload of documentation introduced by Elucidative Programming. With this technique, programmers do not have to worry about the structure of their documents while writing code. The important understanding information is captured through verbalization, and can be structured in later phases when the programmer's workload is reduced. Even if one cannot do both tasks at once, the ability to explain thoughts before or after writing a piece of code, and not being concerned about the structure of documents, their position, and relationship with the source code while programming, is a great achievement in decreasing the workload to an affordable level.

In the next chapter I specify the design and implementation details of the VSCD System.
Chapter 3: VSCD ANALYSIS AND DESIGN

3.1. Introduction

Understanding a program is one of the key requirements throughout a program’s life cycle. Different kinds of understanding information are presented among different people (programmers, analysts, designers, testers) during this life cycle. Although there are many posterior approaches to understanding and interpreting code, programmer’s comments created during the coding process still provide an invaluable window into the rationales and thoughts behind a program that is often not preserved. Several tools and ideas such as Elucidative Programming and elucidators have been introduced to address program understanding at the code level.

In my problem definition I posed the following question:

How can the tool be designed to facilitate the task of documentation by voice?

In this section I present the design of the Verbal Source Code Descriptor (VSCD) tool. The VSCD System follows the basic idea of Elucidative Programming, and introduces an innovative method for collecting and organizing programmer comments. The VSCD data entry system allows programmers to capture their verbalizations using voice recognition as they code, an exercise that keens to facilitate documentation along with the cognitive effort required to generate code. This is similar to the talk-aloud evaluation method commonly used for usability studies [67]. In usability studies, participants are asked to talk aloud their thoughts and tell what they are doing, why they are doing it, and what they think the result will be. This is considered a standard usability technique that is used to find out about a design simply by listening to the tester’s thought process without interfering with the user’s ability to carry out the task [68]. In addition, the data are reliable and accurate [68]. The VSCD System uses the IBM ViaVoice release 9.0 as the recognition engine.

The system also provides a text-based environment that allows users to access standard text-editing formatting features, such as cut, paste, bold, italics, and so forth, that can be used during data entry or for editing during later stages of the project. Another key feature is that documentation and the program exist in separate entities, and users can define relationships between separate pieces of documentation and code to facilitate navigation in both directions.

The final key feature is a new strategy for classifying projects components. This technique classifies all files and components of a program under a new level of abstraction, called a feature. This set of extra levels of classification is designed to provide a better understanding of how a feature is implemented in the source code, especially when the goal is to change or extend the specific feature. Feature understanding is identified as a major aid for program comprehension, and as an
important reverse engineering method for mapping the system’s external behavior to the relevant components of the source code, (see section 2.2.3.1.4.).

These key core features of the VSCD make our research distinct from other tools, such as the Program Elucidator [4].

Since the system is still a prototype, some limitations were placed on the implementation. The first limitation is that the system does not support many of the features and specifications of existing development environments, such as compilation and debugging tools, an object oriented paradigm, and so on. The second limitation is that in the current interface there is only a one to one relationship between features and program components in the classification method. However, this should be expanded to support crosscutting features and also the case where a component is used in the implementation of several features. The third limitation is that the VSCD does not support a multi-user framework at this moment.

To summarize, I decided to focus my project around creating a prototype for verbal documentation. This prototype features aspect of documentation by voice, linking and editing documents and source code, and the question of motivating programmers to use the system is central. I chose not to focus on the feature classification technique at this moment. This will involve further literature studies, design and experiment and I chose to postpone this work until the future.
3.2. Design
In this chapter the design of the VSCD System is presented. First, a description of system's architecture and overview of its major components are provided; then, the details of these components are presented. Some of the design aspects of the Java Elucidator tool [90], especially in the user interface layer, are considered to fulfill the following requirements, as described in Section 2.2.2.2.

1) “The program source file must be intact, without embedded or surrounding documentation” [4, pg. 4].

2) “The programmer must experience support of the program explanation task in the program editing tool” [4, pg. 4].

However, the second requirement also needs the additional support of the voice recognition tool in the VSCD for verbal documentation.

3.2.1. System Architecture
The VSCD System consists of four layers:

1) User interface layer: This provides users with tools and functions to write source code, connect to the voice recognition system and create comments and documentation, and link appropriate pieces of information together for future reference.

2) Functionality layer: This layer provides the core functionality of the VSCD. It consists of four components.
   - Data Manager: This extracts information from the source code and documentation and stores them in the database.
   - Hyperlink Manager: When a user clicks on a hyperlink, the Hyperlink Manager works with a Query Engine to respond to the user's request.
   - Query Engine: This extracts link information from the database.
   - Voice Manager: This communicates with the voice engine to establish connections, provide users with voice recognition results and also terminates the connection upon request.

3) Data layer: This layer consists of all the information used by the VSCD. It is primarily the source files and related documentation. It also contains a database, which is used to store projects, files and link information. These components are described in Section 3.2.4.

4) Voice Recognition layer: This layer includes the voice recognition engine that recognizes verbalizations made by programmers, and converts them to text.

Figure 3.1 illustrates this architecture.
3.2.2. System Configuration

The VSCD is designed using a PC-based environment, mainly due to the availability of the voice recognition application. However, the system concepts are not restricted to one specific platform. Table 3.1 provides details of the environment used to create the VSCD System.

<table>
<thead>
<tr>
<th>System Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operating System</strong></td>
</tr>
<tr>
<td><strong>Programming Language</strong></td>
</tr>
<tr>
<td><strong>Database</strong></td>
</tr>
<tr>
<td><strong>Voice Recognition</strong></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

**Table 3.1**: System Configuration
The Verbal Source Code Descriptor

The above configuration is not necessarily the most efficient one. For example, working with the Visual C++ language would offer more flexibility in choosing speech interfaces, which are more efficient than Microsoft SAPI.

3.2.3. System Components

The internal structure of the Verbal Source Code Descriptor is comprised of various files and components, including a user interface and program files of the system. The VSCD is programmed using Visual Basic 6.0. This section provides a description of the various forms and code modules of the Verbal Source Code Descriptor.

The overall structure consists of one project file called HMPD.VBP and two forms, as described in Table 3.2.

<table>
<thead>
<tr>
<th>Form ID</th>
<th>Form Name</th>
<th>Form Description/Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>frmHLCA</td>
<td>frmHLCA.frm</td>
<td>The main screen of the application that is used to make or open projects, create files and write source code, create comments verbally and edit and link documents. This screen provides users with almost all of the system's tools and functions.</td>
</tr>
<tr>
<td>Findtextform</td>
<td>findtext.frm</td>
<td>Used to find a word or phrase in a document, and if required, replace it with another word or phrase. This form is invoked by the form frmHLCA upon the user's request.</td>
</tr>
</tbody>
</table>

Table 3.2: Forms Descriptions

3.2.4. Data Layer

Source and documentation files are saved in text format and can be read by any text editor, such as MS Word or Notepad.

To keep track of linking information and relationships, a relational MS Access database is defined. The database consists of 2 tables:

Table 1: The LinkInfo table stores all the required information about the address of a linker and the linked objects. Each link object is associated with a file, which is either the source file or the destination file (source and destination can be any source code or comment/documentation). Each link has a unique tag, which is an integer value. The tag associated with the linking object is called Htag, and the one associated with the linked object is called BookMark or BK. Table 3.3 provides the detailed descriptions of the fields associated with each link object.
Verbal Source Code Descriptor

Table 3.3: The structure of LinkInfo table in the database

Table 2: The FilePaths table is used to store information about the source and destination files. A one-to-one relationship between each source code file and its comment/documentation file is defined and specified in this table. Table 3.4 provides the detailed descriptions of the associated fields used in the FilePaths table.

Table 3.4: The structure of FilePaths table in the database

There is a one to many relationship between table FilePaths and LinkInfo, as shown in Figure 3.2.
3.2.5. Voice Engine Layer

IBM ViaVoice release 9.0 is used as the voice recognition engine. Before using the system, the voice engine must be trained by each individual to produce a unique voice model for that individual to a level of reasonable accuracy and performance. It is reported that a product, such as IBM ViaVoice, can recognize 20 to 60 words per minute with an error rate of about 2 to 5 percent once it is trained [70]. The IBM ViaVoice VoiceCenter/User Wizard provides sufficient training facilities for this system. It is also suggested to provide the system with a voice vocabulary that includes words and phrases that are commonly used in programming and documentation, and also specific keywords and literals from the programming language that is in use. This would increase the accuracy level of the voice recognition engine.

Once the system is trained, the programmer can then begin to use the voice recognition system along with an editing environment. The dictated comments are stored in a separate and parallel document file that can be displayed simultaneously with the editor window.

3.2.5. Functionality Layer

This layer provides the core functionality of the VSCD. It consists of four major components.

3.2.5.1. Data Manager

This component stores information about the source code and documentation in the system's depository. The depository consists of the actual source code and documentation files in text format, and also a database including their relationship information, as discussed in Section 3.2.4.
Verbal Source Code Descriptor

The derived information includes the name and path of each file provided by the user through a dialog box and hyperlinks information that is available once it is created by the user.

**Current Implementation**

I created two procedures called, mnuSaveFile_Click and mnuSavePri_Click that are responsible for saving the actual source code and documentation files and creating a configuration file for the project. Hyperlink information is extracted when it is created and defined. Two functions, called Link_Prog_to_Doc and Link_Doc_to_Prog, are invoked when such links are created. At the end of each process the generated information is extracted and stored in the database.

Each hyperlink object has a source and a destination. The source and destination can be either the source code or the documentation. Once a link is defined a value is assigned to each source and destination entities. These values are called the tag value and bookmark value. Hyperlink information thus consists of the source file name, destination file name, tag value and bookmark value. Tag and bookmark values are unique numbers that are considered as ids for the source and destination entities, respectively. The data manager object receives this information and invokes a method called Save_Link_Info to store them in the database.

3.2.5.2. Hyperlink Manager

When a user clicks on a hyperlink, the Hyperlink Manager works with a Query Engine to response to a user’s request.

**Current Implementation**

Two methods SourceWin_Click and DocWin_Click are invoked when a hyperlink object is clicked on in the source code or documentation. As described above, a tag value is associated with each hyperlink object that can be considered as an id for the source entity of each object. Once the object is clicked on, its information is located in the database. This information includes the source file name, destination file name, and tag value. The result is a record that can be used to extract the bookmark value. The bookmark value is used to find the destination of the link.

3.2.5.3. Query Engine

The Query Engine extracts link information from the database.

**Current Implementation**

A method called Query_Engine is responsible for extracting the appropriate information. This method receives the name of the database and the query statement in the form of SQL and executes the query. The result is a record set that consists of all records selected from the database.
3.2.5.4. Voice Manager

Voice Manager communicates with the voice engine to establish connections, provide users with voice recognition results and terminate the connection upon request.

Current Implementation

A method called listen is invoked when the user activates or deactivates the voice engine by clicking on the listen button in the toolbar (see appendix E for details). An object called Vdict1, which is a Visual Basic component, is added to the system to provide an appropriate interface for working with the voice recognition engine. The procedure listen, sets some parameters for activating or deactivating the voice engine. Once the engine is activated, it listens to verbalizations and tries to recognize them. When a word is recognized, a method of Vdict1 object called Vdict1_PhraseFinish is invoked. The recognized word is extracted and placed into the insertion point of the description window.

3.2.6. User Interface Layer

The interface is designed to allow users and/or programmers to control the configuration of the file structure and the voice recognition system, and provides various tools associated with the programming and talk aloud tasks. In this section the interface tools and functions provided by the VSCD are described.

The VSCD implements many of standard MSWindows™ interface tools, such as the open and close functions (see Appendix E for details).
3.2.6.1. Screen Layout and Descriptions

The user begins from the main screen as illustrated in Figure 3.3.

The main screen has 5 key areas. Project Window, Program Window, Description Window, Menu Bar and Tool Bar.

3.2.6.1.1. Project Window

The Project Window displays a hierarchical list of a project and all items contained in the project. Only one project can be opened in this window. The system demonstrates a new way of organizing files and components, based on the features to which they relate. There are two levels of hierarchy in the Project Window. The first level is the list of a project’s features, which specify its behaviour and capabilities. An example of a project feature could be CalcCos, as shown in Figure 3.3. The second level is the list of files under each feature, which specifies the program functional components containing the source code and documentation. An example of a functional component could be the My File.C programming file in Figure 3.3 that consists of functions and procedures for calculating...
the Cosine of angle X. This prototype does not implement crosscutting features, nor the case where a component can be related to more than one feature.

3.2.6.1.2. Program Window

This window appears at the left side of the screen when a file is added to the project. The user types the body of the source code for the selected file in this window. When the user selects another file, the system prompts the user with a message to save the modifications before switching to the other file. The contents of this window are then updated according to the contents of the selected file. Figure 3.3 shows the file *My File.C* selected in Project Window, and the body of this file in the Program Window.

3.2.6.1.3. Description Window

This window appears on the right side of the Program Window. Both the Program and Description windows open when the user adds a new file to the project or selects a file from the Project Window. The dictated text, captured by the voice recognition system, appears in the Description window as the user talks aloud. A description file, which contains the explanations and comments that the user dictates at the time of programming is created and associated with each program file. When the user saves a file, both the program and description files are saved. The relationship and association of the paired program and description file is saved in the system’s database. Whenever a program file is opened, the system finds the associated description file in the database and loads it into the Description Window. The Program and Description windows can be edited and formatted using the edit and format tools of the VSCD System.

3.2.6.1.4. Menu Bar and Tool Bar

A menu-bar is used to allow user access to a file, and to format and voice recognition controls. Many of the menu options appear similarly to many standard software applications. For example, file manipulation commands, such as save and open, edit commands, such as cut and paste, and format commands, such as changing the font and color, are implemented through a menu system and through a standard tool bar. Details of each menu item and tool bar button are provided in Appendix E (the user guide).

3.2.6.2. Verbal Documentation

The voice recognition engine must be activated before verbal documentation. The system provides appropriate tools through the toolbar and menu bar to interact with the voice recognition engine (see Appendix E for more details). There is also a program file called *recplay.exe* associated with the VSCD that can be run to record the programmer’s voice in an audio file format (see Appendix E for details). This capability simplifies the task of editing when the voice recognition engine is not accurate enough, and the programmer cannot remember what was
said originally once there has been a mistake. The voice that is recorded by replay.exe can be listened to by using any standard audio player at the time of editing. To do this, the programmer has to find the recorded voice using a file system, such as Windows Explorer, and play it with the appropriate audio player. Once the voice is playing, the programmer can return back to the VSCD application and continue working with the system. The only problem is that there might be long silence gaps between talks in the recorded audio file that makes it time consuming to listen to. However, audio technology resolved this problem by introducing tools and techniques, such as Ionizer 1.3 introduced by Arboretum Systems Inc. [89], to clean up and compress the sound from noises and unwanted sections.

The system is also capable of recognizing some verbal commands, as described below:

**Go to Sleep**
Prevents the voice system from dictation, but the voice system remains active and waits for the “Wake up” command.

**Wake up**
Allows the voice system to dictate after it has been prevented by the “Go to Sleep” command.

**Select this**
Selects the word under or in front of the cursor.

**Go to Top**
Moves the cursor to the top of the document (Program or Descriptions).

**Go to Bottom**
Moves the cursor to the bottom of the document (Program or Descriptions).

**Cut this**
Cuts the selected text.

**Copy this**
Copies the selected text.

**Paste this**
Paste the clipboard contents in the insertion point.

**Delete this**
Deletes the selected text.

These commands can be expanded further if necessary.

### 3.2.6.3. Editing and Linking

The dictated comments are not usually accurate due to the inaccuracies of the voice recognition algorithms, and some words need correcting. Corrections can be
made by using the mouse and keyboard or by voice commands as the programmer is generating the comments, or at some later time. A number of different types of commands are needed in the VSCD interface. They include commands for moving around (scroll-up, scroll-down, page-up, page-down, top, bottom, left, and right) and editing (highlight, cut, copy, paste, find, and replace). Currently, voice commands are limited to only a few commands of the possible set of voice-based editing comments, such as “Select this”, “Copy this”, “Cut this”, “Paste this”, “Go to Top”, “Go to bottom”, “Go to sleep” and “Wake up”. This limited set is used because these are the most common editing tasks used by programmers, and at this point, in the system’s development, I did not want programmers to spend too much time using voice editing while they were programming. However, as the system develops, more editing and typing commands can be added in future.

Once the programmer is satisfied with his comment set, he can then link those comments to relevant sections of code. The linking task involves selecting appropriate pieces of source code and related comments, and linking them together using a hyperlink structure. There are two types of links available in the VSCD: source to description link and description to source link. Source to description links insert hyperlink objects from the source code to comments and descriptions, and is used when a piece of program is explained in documentation. Description to source links work vice versa and insert hyperlink objects from the comments and descriptions to the source code, and is used when a variable name or function name is mentioned in a piece of documentation and the user wants to see the definition of the variable or function. In Figure 3.3 these links are shown by green or red hyperlink symbols. Clicking on each symbol highlights the related text. For the example in Figure 3.3, the selected symbol is associated with the highlighted text that describes the body of the for-loop. There can also be multiple pieces of code linked to the same comments (or vice versa).

Strategies for determining which piece of source code can link to which section of comments must be developed by the individual programmer. A typical strategy might include relating a section of code to a comment where the particular variable or function name is mentioned. The details of how to work with these tools are explained in the user guide (see Appendix E).
3.3. How to use VSCD

To begin using the VSCD System, the user must either create a new project or open an existing project. Once a new project is created, there can be several levels of abstraction. The first level involves defining various program features to represent functionality and specifications of the given project. Under each feature, related files and components can be added. For example, if a project is required to calculate the salaries of a company’s employees, a feature called “Calculate Net Salary” can be added. The second level then involves files and components of the program, such as GetEmpInfo.C, CalcSalary.C and so on, which are classified under relevant features. The goal of these new levels of abstractions is to improve program understanding by classifying the low-level components in higher descriptive levels and conceptual categories based on the program’s features and specifications.

After defining the project and creating at least one file, it is time to start programming. Programming and documentation can be done together by activating the voice recognition system and explaining the source code while it is being written. Pressing the Listen button on the tool bar, as explained in Appendix E, activates the voice recognition system.

Figure 3.4 illustrates how to use the VSCD step by step.
Verbal Source Code Descriptor

Step 1: A new project is created in the Project Window.

Step 2: A feature is added to the project before creating any file or component.

Step 3: A file is added under a feature and upon creating a file the Program Window and Description Window appear on the main screen.

Step 4: The Voice recognition system is activated by pressing the listen button and prepares the environment to capture user's verbalization.

Step 5: The source code is typed in the Program Window and explanations are captured and converted to text in the Description Window.

Step 6: Errors in dictation are edited using the mouse and keyboard or editing and moving commands.

Step 7: Links between the source code and descriptions are created using the appropriate tools in the toolbar. Each link is specified by a hyper-linked symbol, as shown in Figure 3.3. Clicking on the symbol highlights the specified section, as shown in Figure 3.3.

Step 8: The project is saved along with the program and description files.

Figure 3.4: The process of using VSCD
Chapter 4: VSCD EVALUATION

4.1. Experimental Setup

An initial study was designed to evaluate the influence of the VSCD on a user's experience of programming and documentation. The following research questions were posed as guides for this study:

- Is the Verbal Source Code Descriptor easy to use?
- Whether or not the task of programming and documentation can be done simultaneously?
- How much of the primary evaluation of this technique in terms of quality of comments and workload would be promising for future work?

In this pilot study I mainly focused on documentation by voice, and designed the experiment to address the above questions. The focus of this primary experiment is addressing how to motivate programmers to use the technique of verbal documentation instead of traditional commenting techniques. Therefore, the traditional practice of programming and commenting was considered as a standard for comparison, despite the relationships between the VSCD and Elucidative Programming that might qualify this technique as a better choice for the control group experiment. The main reason to not use Elucidative Programming in my experiment is that this technique is not yet an accepted and known programming discipline among all programmers, and evaluations of this technique found too much overhead due to structuring documentation. Working with Elucidator tools required considerable time and effort that seemed beyond the time limits of my experiment. However, it can be an appropriate comparison index to evaluate the efficiencies of verbal documentation versus Elucidative Programming techniques in terms of workload overhead.

I chose not to focus on the feature classification technique at this moment. This involves further literature studies, design and experiments, and I chose to postpone this work to future.

A subjective workload assessment technique called the NASA Task Load Index (TLX) is used to measure the cognitive workload that the VSCD system imposes on a user, and compare this with the workload of a traditional programming and commenting task. Subjective workload assessment techniques reported to be more popular than other techniques because of their ease of use, non-intrusiveness, low cost, high validity and known sensitivity [84, 87], and among all subjective measurement techniques the SWAT [84] and NASA TLX [85, 87] are shown to be very popular and widely in use [86, 87]. However, SWAT has two main problems: it is not very sensitive for low mental workloads, and it requires time-consuming pre-processing task [87]. The NASA TLX is better than the SWAT in terms of sensitivity, especially for low mental workloads [88, 87]. The details of this technique are discussed in the next section.
4.1.1. Method

Subjects: A sample of 12 participants (six females, six males) was assigned to one of two experimental groups (control group, VSCD group). Six participants were experienced programmers with at least two years of programming experience, and six were undergraduate students in computer science or computer engineering, or had recently graduated from these programs with little programming experience in industry. None of the participants had any previous experience in working with program understanding methods and tools. There was equal representation of gender and programming experience in each group.

Procedure – control group:
Participants in the control group were asked to complete a pre-study questionnaire (see Appendix A) to provide demographic information, and programming and maintenance experience. They were then asked to generate a short program to find the cosine of an angle (X) defined with the infinite sequence \[ \cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots \] using functions for powers \( x^2, x^4 \), up to \( x^{10} \), for factorials (2!, 4!, etc.), and for adding the positive and negative fractions, in an editor that supported their preferred language. This question was taken from a first year final exam. They were also asked specifically to comment their code properly. The verbalizations of the control group participants were not captured and not used in the evaluations. The programming task took approximately 20 minutes.

Following the programming task, participants were asked to evaluate their cognitive workload using the NASA Task Load Index (TLX) technique, a multi-dimensional subjective workload rating technique that has been validated for use [65].

The NASA TLX defines workload as the “cost incurred by a human operator to achieve a specific level of performance” [65, pg. 130]. An overall workload (WWL) score is generated based on a weighted average of ratings of six subscales: Mental Demands (MD), Physical Demands (PD), Temporal Demands (TD), Own Performance (OP), Effort (EF) and Frustration (FR) (see Appendix C for a complete definition of each subscale).

Each participant was asked to read the definition sets and ask questions if needed. Participants were then asked to provide their subjective assessments of the amount of each of the six subscales they experienced upon completing the programming task. A rating scale of 1 to 100, where 1 is the lowest or least of the scale in accordance with the definitions provided was used as required by the TLX instrument. The importance or weights of subscales were specified by a pair-wise comparison. The overall workload was then determined from a weighted combination of scores over the six dimensions.

Procedure – VSCD group:
The VSCD group followed a similar protocol, although they were required to train and use the VSCD System. Specifically, this group was required to: 1) train the voice recognition system; 2) complete a pre survey (Appendix A) to gather demographic data, software programming and maintenance experience; 3) to
complete a post-study survey (see Appendix B) that collected subjective impressions of the VSCD system; 4) to carry out the same programming task as the control group, except that participants were asked to explain their ideas and talk aloud their thoughts as they come while coding their program. A reason for keeping this task short and simple is because first I wanted to see how easy it is to work with the VSCD interface and second, how programmers react to the new discipline of documentation by voice. My approach was to make participants focus on working with the system and do the verbal documentation instead of occupying their time and effort with resolving a complicated programming task; 5) to review and edit the verbalizations made while programming; 6) to link comments to the appropriate points in the participant’s code, and vice versa; and 7) to complete the NASA TLX for the programming, editing and linking tasks. ViaVoice training took approximately of 2 hours to complete. The programming, editing and linking tasks took approximately 20, 25, and 15 minutes, respectively, to complete.

4.2. Results From Observations

This section addresses the following three questions that I posed in Chapter 1 and Section 4.1:

- Is the Verbal Source Code Descriptor easy to use?
- Whether or not the task of programming and documentation can be done simultaneously?
- To which degree is it possible to motivate programmers to do documentation verbally?

How easy is working with the VSCD System?

Before working with the system, each participant was provided with a user instruction guide (see Appendix E). They were given 10 to 15 minutes to review the instructions and practice working with the system before starting the actual experiment. Participants were able to work with the system by using the manual without additional assistance. The results of the post-study questionnaire showed that 4 of 6 participants reported that the system is easy to use. The other 2 participants, however, did not report any difficulty with the system’s interface but frustration caused by the inaccuracy of the voice recognition system and the time consuming task of editing to correct dictation mistakes. The linking process was easy for all the VSCD participants, and 83% (5/6) found this feature useful. In general, participants found this system easy to use. However, a formal usability study is required to further investigate usability problems.

Whether or not the task of programming and documentation can be done simultaneously?

As discussed in Section 2.3, human cognitive ability to do more than one task is very limited. However, in cognitive psychology studies there are evidences of some tasks that can be done simultaneously, such as talking and walking. As a
matter of fact, some tasks that initially required significant cognitive processing are shown to become almost automatic, and therefore, do not require as much attention and focus on the task, for example, typing, CPR, reading, and driving. As these skills are mastered they require less cognitive processing which allows one to be able to do some other things at the same time [83].

The task of programming however, not only involves typing, but also involves a lot of thinking about the solution and its implementation; therefore, it cannot become an automatic task like the simple typing process. Verbal documentation on the other hand, requires attention to concentrate on explanations and describe solutions. Thus, it is assumed that these two tasks cannot be done simultaneously. This means that the process of programming and verbal documentation is serial, not parallel. The observations during the experimental study present some evidence that support this assumption.

Most participants (8/12) reported talking occasionally to often when they write their source code. The first impression of the VSCD System on 50% of the participants was that it was hard to type and talk at the same time, and they preferred to type first and explain later, or vice versa. Although the rest of the participants did not report this difficulty, I noticed that in all of the cases when the subject focused on writing the source code, his or her voice went down and at this moment whatever he or she said did not seem to be useful as a comment (mumblings or repeating the coding syntax). However, the most useful comments were generated when they were not typing and when they were actually thinking about their solutions. The idea of documentation by voice however, has not failed due to this result because the understanding information can still be captured once they are presented.

To which degree is it possible to motivate programmers to do documentation verbally?

Another issue is whether programmers are willing to say their thoughts aloud at all, and how easily they accept this discipline in their daily programming practices. The results of my pilot study showed that although some of the participants were shy in the beginning, they became comfortable with speaking aloud after approximately 5 minutes. 4 of 6 participants found that with some practice they got used to the technique of documentation by voice, but the rest of them (2 of 6) found it very hard to accomplish. Longitude studies in the real world are required to address this issue further.

In general, participants found this system easy to use.

4.3. Data Analysis

Quantitative and qualitative analyses of comments generated by the control group and the VSCD group, and of the subjective mental workload ratings are reported. Comments are analyzed on the basis of quantity of comments generated, and the quality of those comments, as rated by three independent raters. Comments are divided into six standardized comment categories: file header comment, function
header comment, variable definition/id, subsection comment, in-line comment, and separate specification document. These categories are commonly used in teaching commenting techniques. A brief description of each category is presented in Table 4.1. Descriptive results are presented to illustrate the initial differences in commenting between the VSCD and control groups.

Comment quality is assessed using a rating strategy that consists of five quality parameters: depth, explanation, understandability, reasoning, and process order. Table 4.4 provides the definitions of each of the quality parameters. Inter-rater reliability for the rating strategy is assessed using the Interclass Correlation Coefficient (ICC). Statistical analyses are then applied to determine differences in comment quality between the VSCD and control groups. In addition to the results of this analysis, some interesting samples from the observation of the experiment are provided.

Mental workload ratings for the programming task are compared using a t-test statistic between the control and VSCD groups. Answers from the pre and post-study questionnaires are summarized using descriptive and thematic analyses.

4.3.1. Results - Analysis of Quantity of Comments

Traditional commenting techniques introduce 6 standard types of comments that are described in Table 4.1. Based on this classification, counting the comments generated by control group participants seems easy and straightforward. However, the comments and descriptions generated by VSCD participants did not completely fit into this classification framework. There have been some new types of information generated, such as rationales and thoughts behind the solutions, and reasoning that are not usually put in as comments in traditional programming practices. To be able to compare the quantity of comments generated by participants in both groups, I only considered the comments that fit into the standard framework.

The distribution of generated comments and the number of participants generating those comments across the six categories are listed in Table 4.2.
Verbal Source Code Descriptor

<table>
<thead>
<tr>
<th>TYPE OF COMMENTS</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILE HEADER</td>
<td>A comment on top of each file that briefly outlines the contents and purpose of the file</td>
</tr>
<tr>
<td>FUNCTION HEADER</td>
<td>A description of each function, procedure or programming module, above the definition of each module</td>
</tr>
<tr>
<td>VARIABLE DEFINITION/ ID</td>
<td>Definition of local and global variables</td>
</tr>
<tr>
<td>SUBSECTION</td>
<td>A description of each block of code such as loops, If-Else statements and etc.</td>
</tr>
<tr>
<td>IN-LINE COMMENT</td>
<td>A Description of a specific line of code</td>
</tr>
<tr>
<td>SEPARATE SPECIFICATION DOCUMENT</td>
<td>A Description of the overall approach</td>
</tr>
</tbody>
</table>

Table 4.1: Type of comments

<table>
<thead>
<tr>
<th>Type of Comment</th>
<th>Number of participants using specified type</th>
<th>Number of comments</th>
<th>Number of participants using specified type</th>
<th>Number of comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Header Comment</td>
<td>3/6 (50%)</td>
<td>3/53 (5%)</td>
<td>4/6 (66%)</td>
<td>5/108 (4%)</td>
</tr>
<tr>
<td>Function Header Comment</td>
<td>5/6 (83%)</td>
<td>20/53 (37%)</td>
<td>5/6 (83%)</td>
<td>13/108 (12%)</td>
</tr>
<tr>
<td>Variable Definition/ ID</td>
<td>3/6 (50%)</td>
<td>5/53 (9%)</td>
<td>5/6 (83%)</td>
<td>37/108 (34%)</td>
</tr>
<tr>
<td>Subsection Comment</td>
<td>5/6 (83%)</td>
<td>14/53 (26%)</td>
<td>6/6 (100%)</td>
<td>15/108 (13%)</td>
</tr>
<tr>
<td>In-line Comment</td>
<td>5/6 (83%)</td>
<td>11/53 (20%)</td>
<td>6/6 (100%)</td>
<td>38/108 (35%)</td>
</tr>
<tr>
<td>Separate specification document to describe the overall approach.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.2: Number of comments per comment sub-category for control and VSCD groups.

There were a total of 108 comments generated by the VSCD group, compared with 53 in total, generated by the control group for the programming task. This indicates that there were more comments about the experimental program than those generated by control group participants; however, people prefer to say them, as opposed to writing them down, while they are busy with a high cognitive coding task. Although, as discussed in Section 4.2, most participants did not talk
and type simultaneously, when they stopped typing to think about their solutions, they had more motivation to explain their code as part of their thoughts, rather than writing comments in the source code that requires additional attention to think about where to place them and what to write. This way, they had more freedom in saying whatever they thought, and whenever it is more convenient. Editing and linking processes then would give them enough time to elaborate and clean up the descriptions, and make appropriate relationships. As explained before, control group participants were explicitly asked to comment their code properly, and the VSCD participants were also asked to say their thoughts aloud. There was not any specific condition causing the VSCD group to generate more comments, and it is assumed that participants understood the request for talking aloud their thoughts, and they did not misinterpret this with generating more comments.

One of the key questions in the pre-study questionnaire asked participants to rate, on a scale of one to seven, the importance of the six comment sub-categories listed in Table 4.2 from a programming or maintenance point of view. It is assumed that only those participants who had maintenance experience (8 out of 12) considered this question from both maintenance and programming point of view, and the rest (3 out of 12) only considered the programming aspects of commenting, such as readability. One participant did not answer the rating question. Ratings of the most or second-most level of importance (rating of 1 or 2) and the least or second-least level of importance (rating of 6 or 7) for each sub-category of comment are listed in Table 4.3.

A comparison of the participant’s subjective ratings of comment-type importance with the actual types of comments that were generated during the coding exercise in two different disciplines provides interesting results, which are discussed below.

<table>
<thead>
<tr>
<th>Type of Comment</th>
<th>Participants with maintenance experience</th>
<th>Participants without maintenance experience</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># of ratings as 1st or 2nd in importance</td>
<td># of ratings as last or second last</td>
</tr>
<tr>
<td>File header comment</td>
<td>4/8 (50%)</td>
<td>1/8 (12%)</td>
</tr>
<tr>
<td>Function header comment</td>
<td>4/8 (50%)</td>
<td>0</td>
</tr>
<tr>
<td>Variable definition/ ID</td>
<td>0</td>
<td>8/8 (100%)</td>
</tr>
<tr>
<td>Subsection comment</td>
<td>3/8 (37%)</td>
<td>2/8 (25%)</td>
</tr>
<tr>
<td>In-line comment</td>
<td>4/8 (50%)</td>
<td>0</td>
</tr>
<tr>
<td>Separate specification document</td>
<td>0</td>
<td>5/8 (62%)</td>
</tr>
</tbody>
</table>

Table 4.3: Ratings of importance by comment sub-category
The data summarized in Tables 4.2 and 4.3 show that there were no comments generated as a separate specification document as the programming task in our study did not require such document. Most participants (5 of 8 with maintenance experience, and all participants without maintenance experience) did consider this type of document a low priority, and it was not evaluated in this study.

The most common types of generated comments by the control group are function header comments, 20/53 or 37% of comments, compared with only 13/108 or 12% in the VCSD experiment. In addition, most participants in the control and VSCD groups (83%) generated function header comments. The results of ratings for this category show that most participants (4 of 8 with maintenance experience, and 2 of 3 without maintenance experience) believed that this type of comment is important. However, the most common types of comments generated by the VSCD group were in-line comments (38/108 or 35% for VSCD compared with 11/53 or 20% for the control group), and all of them generated comments of this type. The results of the ratings for this category are the same as the function header results, and show the same importance level.

Comparison of results indicates that the VSCD experiment subjects mainly started by explaining the details of solutions corresponding to lines of code, instead of giving a more general overview of the program first. On the other hand, control group participants preferred to use more high-level and functionally descriptive comments instead of low-level comments that describe lines of code for the sake of readability. Because both types of comments were reported as important, a good discipline must facilitate generating both types. The VSCD System resolves the problem with in-line comments, but it should provide a template to structure descriptions from a general level to specific details at the editing phase.

Pre-study ratings of importance for variable definition comments showed that this type of comment is the least important comment among most participants (8 of 8 participants with maintenance experience, and 2 of 3 without this experience); however, 34% of comments generated by the VSCD System were of this type, which is a considerable amount in comparison to other types. The value and usefulness of having extensive variable definitions for program maintenance remains to be evaluated in longitudinal studies.

In four of the six comment categories more than 80% of the VSCD participants generated comments, whereas in the control group there were only three categories where more than 80% of the participants generated comments. This may indicate that most programmers have comments to make in a majority of the standardized comment categories because they can verbalize them, but that they do not regularly type them as comments in code.

4.3.2. Results - Quality of Comments

The quality of comments can be defined based on the amount of understanding information that they provide. However, measuring the understandability needs a tool to define a set of factors that would contribute to a better understanding of the source code. The literature review demonstrates that most work in the area of
program understanding focus on tools and techniques for providing a variety of "understanding information"; however, no work has been done to compare and evaluate each information in terms of the level of contributions to understandability for defining the appropriate measurement tool. I defined a rating procedure with 4 sub-scales (Depth, Explanation, Reasoning, Process order) based on a hypothesis that these factors provide the most important understanding information. The definitions of these factors are shown in Table 4.4.

<table>
<thead>
<tr>
<th>RATING SCALE DEFINITIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
</tr>
<tr>
<td>---------------</td>
</tr>
<tr>
<td>DEPTH</td>
</tr>
<tr>
<td>EXPLANATION</td>
</tr>
<tr>
<td>REASONING</td>
</tr>
<tr>
<td>PROCESS ORDER</td>
</tr>
</tbody>
</table>

Table 4.4: Comment's quality rating scales definition

Three raters chosen from people with some programming background who were not aware of the experimental task, rated the quality of comments produced by the 12 subjects (six control group subjects and six VSCD group subjects) using these sub-scales. Ratings were from 1 to 10. An Interclass Correlation Coefficient (ICC) was performed between the 3 raters for each sub-scale. The inter-rater reliability was good for explanation (ICC = 0.69, p<0.05), moderate for depth (ICC = 0.59, p<0.05), and weaker for process order (ICC = 0.40) and reasoning (ICC= 0.4).

An independent samples t-test was carried out on all the quality comment categories between the control and VSCD groups to determine if there were any differences in the quality of the comments generated in the two groups (see Appendix D for the t-test results tables). There was a significant difference between the VSCD group and the control group in the two categories of quality: depth [t=2.57, p<0.05], and reasoning [t=3.97, p < 0.05]. The other categories showed no difference in quality between the VSCD and control groups. The
means and standard deviations for all quality categories by group are shown in Tables 4.5.

<table>
<thead>
<tr>
<th>Category</th>
<th>Control Group</th>
<th>VCSD Group</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>SD</td>
</tr>
<tr>
<td>Depth</td>
<td>4.00</td>
<td>1.30</td>
</tr>
<tr>
<td>Reasoning</td>
<td>1.92</td>
<td>0.58</td>
</tr>
<tr>
<td>Explanation</td>
<td>6.25</td>
<td>2.19</td>
</tr>
<tr>
<td>Understandability</td>
<td>6.75</td>
<td>1.81</td>
</tr>
<tr>
<td>Process Order</td>
<td>8.17</td>
<td>2.14</td>
</tr>
</tbody>
</table>

Table 4.5: Mean and standard deviation for quality categories

In addition to these results, some interesting comments, generated by the VSCD group, show the differences in the information provided by both groups. Below are some examples of programs and relevant descriptions provided by VSCD participants. A complete set of all comments generated by VSCD participants are provided in Appendix H.

<table>
<thead>
<tr>
<th>Source Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*Import java.Math;* *Import java.lang;*</td>
<td>Importing some useful libraries such as math and java.lang, these \textit{will help us print on the console}. (Describes why he used these Java libraries)</td>
</tr>
<tr>
<td>\textbf{For } (int \textbf{I=0;} \textbf{I&lt;arrValue.length;I++}){ System.out.println(...) }</td>
<td>This print statement is going to be in a for loop which \textit{will iterate the \textbf{length} of the array}. (Explains the purpose of the method \textbf{length} of the array object \textbf{arrValue})</td>
</tr>
<tr>
<td>*Angle[0]=0;* *Angle[1]=pi/6;* *Angle[2]=pi/5; *...</td>
<td>I need to write a class that has three main methods, one for calculating factorials, one for calculating powers, and another for actually generating the cosine of the angle. (Describes his thought process).</td>
</tr>
</tbody>
</table>

\*Angle[0]=0;\* \*Angle[1]=pi/6;\* \*Angle[2]=pi/5; \*... | I need to store the value of each of the angles. The first one is zero, the next one is pi divided by six and so on. I think I’ll put this into a for loop and save myself some typing. Increment the angle number and do the division. No the for loop was too much work, so I just calculated them one by one... (Describes his thoughts and why he did not choose a solution based on... |
Verbal Source Code Descriptor

Table 4.6: Samples of source code descriptions generated by VSCD participants

<table>
<thead>
<tr>
<th>Verbal Source Code Descriptor</th>
<th>his original thoughts</th>
</tr>
</thead>
<tbody>
<tr>
<td>add = FALSE</td>
<td>I need to figure out if I’m adding or subtracting, so let’s define a Boolean variable for that—I’ll call it add—if it’s true we add, if it’s false we subtract. The first time I want to subtract so I’ll initialize it to false. (Describes his problem and how he implemented his solution.)</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>if (add)</td>
<td></td>
</tr>
<tr>
<td>output = output + frac;</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>output = output - frac;</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

As we can see in the above descriptions, there is some information that is not usually written as comments in the source code. Some of the above descriptions introduce reasoning behind the solutions that are not usually written as comments. It is expected that as a program becomes more complicated more useful understanding information can be generated.

4.3.3. Subjective Workload Measurement Results

Following the completion of the programming task, all participants were asked to complete the NASA TLX assessment. The goal was to measure and compare the workload demands on a user introduced by the commenting tasks in each experimental group. Independent sample t-tests were performed to compare the control group and the VSCD group for the six sub-scales (as defined in Appendix C) and the overall workload measures. There were no significant differences in mental workload for any of the six sub-scales, and overall workload measures. Table 4.7 provides the mean and standard deviation for all of the workload sub-categories and the total workload.

It is assumed that programmers’ capacity for extra overhead due to documentation is limited so the amount of time and effort that they usually spend on documentation is what they can afford, and these never exceeds their capability. Therefore, the result of not getting significant differences in mental workload may indicate that documentation by voice was able to keep a programmer’s workload at a reasonable and affordable level, and yet provide more useful documents than traditional programming practices.
Table 4.7: Mean and SD for mental workload sub-categories ratings and overall workload

Figure 4.1 shows the comparison of the means of each mental workload sub-category and the overall workload between two experimental groups. Figure 4.1:

Mean for mental workload sub-scales and overall workload (WWL)

The diagram shows that there is a trend toward a positive impact of the VSCD group. For example, there is considerable improvement in perception of own performance for the VSCD group over the control group (ratings of 40.8 compared with 24.1 respectively). However, frustration levels are higher for the VSCD group (rating of 37.5 compared with 16.7). Although frustration was rated very high by 50% of VSCD participants, it was not given a high weight for its contribution in the workload by 66% of participants, including those with high ratings. The Mental Demand and Own Performance gained the highest weights. (The mean of weightings for Mental Demand was 0.24 whereas Frustration weighted 0.18 on average).
Observations confirm that the VSCD experienced higher frustration due to the voice recognition system. Most participants were not that successful using ViaVoice in such a short duration, and had difficulties attaining the appropriate accuracy. However, one of the participants had previous work experience with ViaVoice and was able to attain reasonable accuracy. His frustration level was lower than any other subject. It is reported that as users spend more time training ViaVoice and gaining experience using it, they can improve their accuracy level and be less frustrated with the system [70].

4.4. Summary

The results of the initial study suggest that the VSCD has advantages over the traditional or typical commenting approach of typing in comments during or after the coding process, particularly in the areas of comment quantity and quality. There is also a positive trend in mental workload ratings for the VSCD system that supports positive reactions to the VSCD on questionnaires.

The number of comments generated by the VSCD group was almost twice the number of comments generated by the control group (108 vs. 53). Programmers seem to have more explanations and comments to say about their source code than what they actually would type as comments in their source code. This indicates the use of the VSCD may encourage programmers to provide more detailed explanations for their source code using speech.

As mentioned in Section 4.3.1, results of quantity analysis of comments generated by the control group show that these participants commonly used function header comments. This indicates that programmers in this group preferred to use high-level descriptions in the form of function and sub-routine headers instead of in-line and detailed descriptions in the body of their code.

However, the results obtained from the similar analysis on the VSCD group showed that the stream of a programmer's consciousness captured by the voice recognition system is in-line comments. In-line comments may also have been more abundant with the VSCD group because comments were being added to a separate document file and there was no concern with program readability.

Other quantity results are more difficult to explain. For example, variable definition comments comprised a high number of generated comments by VSCD participants, but were rated as not important by most of the participants (10/11). Further research on actual commenting practices and behavior is required to understand the relevant comment types and their effectiveness in supporting program maintenance.

Regardless of the usefulness of each of these comment categories, more than 80% of VSCD participants generated comments in more categories than the control group participants. This again supports the one advantage that VSCD has over the traditional typing of comments in that more comments and in-depth explanations about a program can be captured when it is developed without having a significant impact on the programmer’s overhead.
Verbal Source Code Descriptor

The results of the analysis of comment quality showed a significant difference in depth and reasoning. Comments generated by the VSCD group were longer and had more depth. Participants from this group tended to provide lengthy explanations as they occurred, while control group programmers entered a majority of their comments after completing the programming task. Some VSCD generated comments even captured a revision thinking process in detail so that there was an understanding of the reasoning behind how and why a programmer revised a piece of code to resolve a problem. Comments of how programmers came up with their ideas were also captured at the time of programming providing insight into the programming process and mindset of these individuals. This is then expected to increase a program’s understandability because the reasoning and decision-making process of the original programmer is explicit. Further research is required with the original programmers and program maintainers to verify this hypothesis and explore the extent of program understandability resulting from this type of comment or information. While there are some positive trends in the other quality factors, additional research is required to explore the extent of this trend, and the impact that other factors would have on program maintenance.

There was no significant difference in the mental workload between the VSCD and control groups. This result may also indicate that there is no increase or decrease in perceived cognitive workload as a result of the voice recognition system or the talk aloud task. Further, research with more subjects and more complex programming tasks over longer periods may provide a better indication of the true differences in the mental workload experienced by programmers using the VSCD system and traditional commenting techniques. In addition, a study that examines the mental workload of program maintainers who have access to VSCD comments versus traditional comments may also provide insight into the advantages and disadvantages of the VSCD system.

The observations of participants while working with the VSCD System support some of our expectations. Although some participants indicated that they do not talk aloud when they write their source code, all participants in the VSCD became comfortable with speaking aloud after approximately 5 minutes. The cognitive process of each task (programming and verbal documentation) seems considerably high; therefore, it is expected to not be able to do both tasks simultaneously. The results from observations and also reports gathered through questionnaires confirmed this assumption. It was observed that the most useful understanding information was present when programmers were not typing the code.

The recognition errors introduced by weak accuracy rates for the voice recognition system probably caused more disruption and frustration than desired. Participants had to pay attention to the words (text) being generated, which was more distracting and frustrating than expected. One participant did have considerable experience with voice recognition (daily use of voice recognition) and did not experience the same level of frustration and disruption because his accuracy levels were very high. As programmers gain experience with a specific voice recognition system, the system learns the unique characteristics of an
individual's speech and becomes highly accurate. It is expected then that frustration and disruption levels for programmers would decrease over time as they use the voice recognition system more.

Because of the number of errors in the recognized speech, editing task took a considerable amount of time and effort. Sometimes it was difficult for people to decipher the text of what they originally had said, and they had to listen to their recorded voice to accomplish the editing and linking tasks. Again, as programmers gain more experience and practice with the voice recognition system, accuracy levels improve and the readability of the text increases. Longitudinal studies are required to determine how much experience is required and whether some of these issues are actually resolved.

The NASA TLX mental workload ratings indicated that linking process was the easiest part of the experiment. In addition, all participants found this feature very useful. Normark [4] also reported that a linking system is an important addition to prevenient programming systems.
Chapter 5: CONCLUSION AND FUTURE WORK

In this report I presented a summary of existing techniques, ideas and approaches in the literature of program comprehension. As discussed in the section of program understanding classification, these techniques are divided into two major categories, posterior and prevenient. According my investigation, most of the research so far, focused on posterior approaches that try to extract information from a program after its development phase and at the time of maintenance. Most of the implemented software tools for the purpose of program understanding use various static and dynamic analysis techniques to reconstruct the design and implementation information that is missing in documentation. However, this effort takes considerable time and is expensive.

Some researchers, such as Normark [4] and his students, focus on prevenient approaches that are trying to capture the understanding information from original developers at the time of program development. This approach provides a structured framework for documentation and its goal is to motivate programmers to produce high quality documentations at the phase of implementation. However, the idea of capturing understanding information from programmers in such a structured format has a bottleneck, which is the "Programmer’s Mental Overheads" that causes these approaches to not be accepted by programmers in real practices.

In my research work I examined an alternative input method for capturing the required information with less overhead. I focused on gathering verbal information using voice recognition methods, and analyzing the information to link them to the appropriate pieces of source code. The fact that most programmers talk to themselves while programming was the initial motivation for pursuing this idea. This approach enables programmers to do the task of writing the code and documentation with less effort than Elucidative Programming. This strategy offers two major benefits to programmers: first, it gives them enough freedom to talk aloud their thoughts without being restricted to a specific format; and second, talking while thinking takes less effort than typing.

I developed a prototype tool with the feature of voice recognition to capture and record the verbal descriptions. Programmers are able to verify this information before saving them in the system’s database. The system provides some tools and facilities to link the source code to the appropriate pieces of documentation.

I also proposed a higher level of abstraction in the development environment than classes, modules, functions and procedures. This abstraction is based on various conceptual aspects and features of the program. For example, if a program has a feature to provide a specific report then the programmer should add this aspect to the project and then add relevant modules and functions underneath this class.

Thus, when you open a project at the first level of abstraction you can see all the features and aspects of the program, and under each aspect you can simply find all
relevant modules and components. This approach could be a resolution for the problem of localizing the implementation of features in the code, as discussed earlier in this report. The evaluation and analysis of this proposal is suggested for future work.

After the development of this system I conducted an experimental study with a small sample of users, consisting of 12 participants divided in two groups: the VSCD group and the Control group. The experiment was relatively long (almost 3 hours) and some participants had to return 2 or 3 times to complete the experiment. Therefore, it was difficult to recruit voluntary subjects (minimal compensation for participation due to ethics requirements). For future experiments, especially the longitudinal studies for analyzing the usefulness of VSCD comments for program maintenance, I would suggest that a partnership with a software company be formed where employees can be allowed to participate during their work hours.

The following is a summary of the results obtained from this study:

- Observations and questionnaires showed that participants found the VSCD System easy to use. However, a usability study is required to further investigate usability problems.
- Fifty percent of participants reported that they cannot talk and program at the same time and observations of the rest of the participants showed that they are almost in the same situation. This suggests that the task of writing the code and verbal documentation are serial tasks.
- Most of the participants (4 of 6) found that with some practice they can get used to the technique of documentation by voice, but the rest (2 of 6) found this very hard to accomplish.
- Patterns of commenting and quantity of comments were different between the two groups.
- There were significant differences in Depth and Reasoning aspects of comments generated by both groups. The comments generated by the VSCD group posed more depth and reasoning.
- There were no significant differences in mental workload results obtained from the control group and VSCD group. This indicates that the verbal documentation workload is not beyond human capability.
- Workload measurement results showed a higher frustration level for VSCD participants in comparison to the control group. This frustration was likely caused by the inaccuracy of the voice recognition system. Cognitive science researchers have reported that current voice recognition systems such as Dragon Naturally Speaking and IBM ViaVoice, recognize continuous natural language speech at 20 to 60 words per minute with an error rate of about 2 to 5 percent [70]. It is also expected that these technologies would be improved over time. The lack of time in my experiment prevented participants from training the voice recognition
engine properly. However, one of the participants with previous ViaVoice experience showed better performance. Although the inaccuracy of the voice recognition system can be improved by training over time, the usefulness and necessity of voice recognition in the VSCD System should be investigated further. On one hand, the use of voice recognition at an accurate level would simplify the task of converting voice to text and considerably save a programmer’s time in typing, on another hand, considerable time (6 months or more) is required to accomplish the training phase, and at that point voice recognition accuracy still depends on other factors, such as the clarity of speaking and the tone of voice. Even at a very high accuracy level, the voice recognition system expects people to talk clearly and word-by-word, which is not how people naturally speak, especially when they are thinking their voice goes down and cannot be easily recognized. Speaking to a voice recognition system in a recognizable way requires extra effort that would fail one of my major goals, which is to decrease the overhead of documentation. Therefore, further investigation is required to specify the usefulness of converting voice to text, and if it is not applicable, substitute voice recognition with alternative techniques such as recording the voice in acoustic wave form and provide users with tools to analyze the voice. This feature is similar to inserting voice comments in WORD documents.

In general, it appears that the VSCD System facilitates the recording of programmer comments without adding any extra mental workload or effort to the programming task. Longitudinal and more extensive studies are still required to determine the usefulness of the VSCD generated comments for program maintainers.

The following suggestions are recommended and proposed to be done in future:

1. Some changes in the technology in use are required to improve the efficiency of the system. Examples of such changes include the programming language (Visual C++ is suggested instead of Visual Basic), and the speech development interface (SMAPI or ViaVoice SDK instead of Microsoft SAPI).

2. Building automatic program-to-program, program-to-documentation or documentation-to-program relationships by applying an appropriate parser.

3. Expanding the tool to support a multi-user paradigm.

4. Designing the feature-oriented classification technique to support crosscutting features and aspects.

5. Applying appropriate usability study techniques on the system to find and recover its usability problems.

6. Investigating and examining the application of the voice recognition system in verbal documentation and verifying its usefulness.
In addition to some developments in the application, the system has to be evaluated and studied in a larger domain and scale with some real world problems. The following hypothesis could be used as a guide for such studies:

- **Hypothesis 1:** A program that is written in a verbal documentation paradigm provides more understanding information than other techniques (Posterior), or complements the understanding information provided by such techniques.

- **Hypothesis 2:** The mental workload of verbal documentation is affordable and acceptable by programmers in comparison to the Elucidative Programming technique.

- **Hypothesis 3:** Programmers can accomplish feature-oriented classification at the phase of implementation, and this increases program understanding at the time of maintenance.

- **Hypothesis 4:** The system would scale up to a complex programming task with many modules.

To test hypotheses 1, 3 and 4, two groups of programmers will be assigned to resolve a complicated problem, one group using VSCD and the other working with common programming development environments. Each programmer using VSCD must be given at least a two-day tutorial on the use of voice recognition software to ensure that each is fully trained in its use. The use of the voice recognition system must be followed for a period of 3 months. The three-month training insures that all participants are proficient in the use of voice recognition.

After this phase, two groups of maintainers will be asked to fix a bug that is already applied on each program, one group using the verbal documents and the other using reverse engineering and posterior techniques. Then, the results of their performance can be analyzed and evaluated. To test hypothesis 2, programmers will be divided into two groups; half will work with VSCD, and the other half will work with Elucidator, and their workload will be measured and compared. The suggested workload measurement technique is NASA TLX; however, there are other measurement techniques that might seem appropriate [65]. Tests will be done in two different environments: industrial and academic environments to compare the differences between the performance of experienced programmers and maintainers with students.

This research has taken some initial steps towards using alternative input techniques to improve the commenting process during programming tasks by taking advantages of multimedia techniques such as voice recognition and audio processing. The positive results of the research suggest that further exploration, study and development of these techniques would be worthwhile and potentially have a positive impact on comment quality and quantity.
Appendix A: Pre-Study Questionnaire

This questionnaire applies to our study assessing the use of voice recognition to capture computer program comments for software development. The purpose of this questionnaire is to gather personal information as well as current software programming, software maintenance and documentation practices. It takes approximately 10 minutes to complete this questionnaire. Thank you in advance for your participation in our research.

Section 1: Personal Information:

1. Please indicate your age by checking the appropriate box.

[ ] Under 20
[ ] 20 - 29
[ ] 30 - 39
[ ] 40 - 49
[ ] 50 +

2. What is your gender?
[ ] Male
[ ] Female

3. What is your current employment status?
[ ] Student  [ ] Employed  [ ] Other (Please Explain)

4. What level of education have you completed?
[ ] High School Diploma
[ ] College Certificate
[ ] Undergraduate 1st and 2nd year
[ ] Undergraduate 3rd and 4th year
[ ] Graduate Degree (Master and Ph.D.)

5. What is your main field of study?
[ ] Computer Science
[ ] Software engineering
[ ] College level programming languages or operating systems
[ ] Information technology
[ ] Other, please specify

6. What is your occupation?
[ ] Computer Programmer/Developer
[ ] Software engineer
Verbal Source Code Descriptor

[ ] System Analyst
[ ] IT Manager
[ ] Other (Please specify) ________________________________
[ ] Not applicable

Section 2: Software Programming Experiences:

7. Please indicate your experience as a computer programmer. Please check one.
[ ] No experience
[ ] Summer or Co-op employment
[ ] Entry Level (less than one year of work experience)
[ ] Intermediate (1 – 3 years working experience)
[ ] Expert (Over 3 years working experience)

8. What programming languages do you usually use? Check all that apply
[ ] Java, J++, JavaScript  [ ] C/C++
[ ] Visual Basic  [ ] C#
[ ] Visual C++  [ ] FoxPro/ Visual FoxPro
[ ] Delphi  [ ] Pascal
[ ] Other (Please specify) ________________________________

9. How often do you place comments inside your source code to describe your code?
[ ] Always  [ ] Very Often  [ ] Often  [ ] Occasionally  [ ] Seldom  [ ] Never

10. What is your most common style of commenting?
[ ] I put in lots of comments at the beginning of a program, but less as the program progresses.
[ ] I put a consistent amount of comments throughout the program while coding.
[ ] I put most of the comments in after programming is complete.

11. Your commenting style: Rank, from 1 to 7, the level of importance that you place on each item in the following list. 1 is most important; 7 is least important. Use each number only once.

[ ] File header comment (A comment on top of each file that briefly outlines the contents and purpose of the file.)
[ ] Function header comments (A description of what each function, procedure or programming module does, immediately above the definition of each module)
[ ] Define local variables within a function
[ ] Subsection comments that describe each block of code, such as loops, If-Else statements, and so forth, within a function
[ ] In-Line comments to describe a specific line of code
Verbal Source Code Descriptor

[ ] Separate the specification document to describe the overall approach
[ ] Other (Please explain)

12. Do you talk out loud to yourself when you program?
[ ] Always
[ ] Very Often
[ ] Often
[ ] Occasionally
[ ] Seldom
[ ] Never
[ ] Don’t Know

13. Some programmers do not like to be interrupted by documenting and
commenting their source code while programming. Others do not mind being
interrupted. If a software tool could reduce the interruption caused by
switching between documentation and programming tasks, rate how helpful
this tool would be for you.

[ ] Not helpful at all [ ] Not that helpful [ ] Somewhat helpful [ ] Helpful
[ ] Very helpful

Please explain why you would find this type of tool helpful

14. Have you ever worked on someone else’s source code to make alterations,
enhancements or bug fixes?
[ ] Yes [ ] No

If you checked “No” please return the questionnaire now-Thank you for your
time. If you checked “Yes” please proceed to Section 3.

Section 3: Software Maintenance Experiences:

15. Think of your recent maintenance activity; please briefly explain what you had
to do.

16. Rate the usefulness of the original programmer’s comments.
[ ] Not available [ ] Useless [ ] Somewhat Useful [ ] Very Useful
17. The original programmer's comments were:
[ ] Too few       [ ] Enough       [ ] Too many

18. The original programmer's comments were:
[ ] In the wrong place
[ ] Sometimes in the right location, sometimes in the wrong location
[ ] Always in the right location

19. Rate the usefulness of the specification documents
[ ] Not available [ ] Useless       [ ] Somewhat Useful [ ] Very Useful

20. What difficulties have you experienced when trying to maintain software?
(Check the one that caused the most difficulty.)
[ ] Understanding the existing comments.
[ ] Following the program flow.
[ ] Tracking variables, constants and function names.
[ ] Unavailable or insufficient explanations of the code
[ ] Missing context (I didn’t have a good overall picture of the system’s features and functionality)
[ ] Other (Please specify)

21. Which of the following program understanding tools you have used?
[ ] CodeSurfer       [ ] HyperCode
[ ] CC-Rider         [ ] IMPACT
[ ] Exuberant Ctags  [ ] VBSurfer
[ ] Imagix 4D        [ ] Project Analyzer
[ ] Java2HTML        [ ] Other
[ ] I have never used a program understanding tool

22. If the original programmer were not available to you when maintaining a program, what other tools would you find most useful to assist understanding of the source code? Select of the following three options that you would find most useful and rank from 1 to 3, where 1 is the most useful tool, and 3 is the least useful tool.

[ ] Hyperlinks where source code is converted into color-coded hyperlinks that connect function calls and variables to corresponding definitions or Hyperlinks of references within comments.
[ ] Control flow analysis. (The control of basic blocks of the program.)
[ ] Program dependency analysis where the dependency among various parts of the source code is analysed, and how a part of a program or a variable can effect
other parts of the program, or which parts will be affected by a variable or program point, is specified)
[ ] Automatic documentation generation where comments embedded in the code are extracted and placed in a separate formatted document. Note: the programmer must generate the comments within the code.
[ ] Feature analysis where features of the program can be followed as the program is executed.
[ ] More detailed comments from the original programmer's thoughts.

23. Please specify any other features, functions or activities that you find particularly useful when maintaining a program? (Please explain)

24. Please specify any other features, functions or activities that you find particularly useless or distracting when maintaining a program? (Please explain)

25. Have you ever worked with voice recognition applications?
[ ] Yes        [ ] No

Thank you very much for taking the time to complete our survey.
Appendix B: Post-Study Questionnaire

We would like to gather your overall impression about the features and specifications of Verbal Source Code Descriptor (VSCD) software that you have just experienced. Thank you in advance for your assistance with our research.

Section 1: Feature Based Classification Approach

You were introduced to a new level of abstraction in the project that you have designed based on the features and functionality of your project. This introduces a new way of classifying your project. In this section we would like to ask your opinions about this classification.

1. This classification would help people understand the project better at the time of maintenance. (Check one)

[ ] Strongly Agree  [ ] Agree  [ ] Somewhat Agree  [ ] Disagree  [ ] Strongly Disagree

Please explain your selection.

2. Rate your level of difficulty with doing this classification: (check one)

[ ] Very difficult  [ ] Difficult  [ ] Somewhat difficult  [ ] Easy  [ ] Very Easy

3. What difficulties did you find while you were doing classification? (Check all that apply)

[ ] Some files and modules cannot be classified.
[ ] Features are shared by more than one file and/or modules.
[ ] It is difficult to keep track of everything.
[ ] It was difficult to learn how to do the classification.
[ ] No difficulty.
[ ] Other (Please specify)

4. What were the positive aspects of this classification approach? (Check all that apply)

[ ] It is easy to keep track of all of the program components using this approach.
[ ] It was easy to use to organize my code.
[ ] It was easy to learn how to use the system.
5. Please indicate the advantages for code maintainers using this classification approach. (Check one)

[ ] It helps in understanding what the project is capable of doing in a glance.
[ ] It helps in understanding the purpose of each program component and its relationship with the system’s specifications.
[ ] No advantage.
[ ] Other (please specify)

6. Please indicate the disadvantages for code maintainers using this classification system. (Check one)

[ ] It would not be useful to people who would be responsible for maintaining my code.
[ ] The system makes it more difficult for programmers to understand the purpose of each program component and its relationship with the system’s specifications.
[ ] No disadvantages
[ ] Other (please specify)

Section 2: Verbal Commenting

The system allows programmers to type source code while capturing verbal explanations of that code at the same time. In this section we would like to know your opinion about this feature.

7. The process of typing the source code and verbally describing it at the same time was: (Check one)

[ ] Very Hard  [ ] Hard  [ ] Not Hard but needs practice  [ ] Easy  [ ] Very Easy

8. Which sentence best describes your first impressions of writing code and explaining it at the same time? (Check one)

[ ] It was confusing.
[ ] It was fast to write source code and document at the same time.
[ ] I could not type and talk at the same time. I prefer to type first and explain later or vice versa.
[ ] I prefer to type my explanations instead of talking
[ ] Other (Please Specify)
9. Do you think that explaining source code verbally using VSCD would decrease the programmer’s overhead due to documentation and commenting? (Check one)
[ ] Strongly Agree [ ] Agree [ ] Somewhat Agree [ ] Disagree [ ] Strongly Disagree

Please explain your selection

10. The amount of editing required to correct your documentation after dictating was: (check one)
[ ] Very much [ ] Moderate [ ] Not much [ ] No editing

Section 3: Linking the Source Code and Documentation to each other

The system supports the ability to connect your source code to relevant parts in the documentation and vice versa. In this section we would like to know your opinion about this feature.

11. The process of linking is: (Check one)
[ ] Very hard to remember [ ] Hard [ ] Not hard but need practice [ ] Easy [ ] Very easy

12. Hyperlinking the source code and documents to each other is: (check one)
[ ] Useful [ ] Somewhat useful [ ] Not useful

Please explain your selection.

13. Separating the source code and comments into separate files and windows and linking them together
[ ] improves source code readability and consequently program understanding.
[ ] improves source code readability but does not improve program understanding.
[ ] does not improve source code readability but improves program understanding
[ ] neither improves source code readability nor program understanding
[ ] Other (Please explain)
Section 4: General Questions

14. In general, rate the ease of use of the system:
[ ] Very easy  [ ] Easy  [ ] Somewhat Difficult  [ ] Difficult  [ ] Very Difficult

15. If difficult, please explain the difficulties briefly.

____________________________________________________________________________

____________________________________________________________________________

____________________________________________________________________________

16. Would you buy such software for the purpose of program development and maintenance?
[ ] Definitely  [ ] Not sure at this moment  [ ] Never

Thank you very much for taking the time to complete our study.
Appendix C: NASA Task Load Index (Scale Definitions)

<table>
<thead>
<tr>
<th>Title</th>
<th>Endpoints</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>MENTAL DEMAND</td>
<td>Low/High</td>
<td>How much mental and perceptual activity was required (e.g., thinking, deciding, calculating, remembering, looking, searching, etc.)? Was the task easy or demanding, simple or complex, exacting or forgiving?</td>
</tr>
<tr>
<td>PHYSICAL DEMAND</td>
<td>Low/High</td>
<td>How much physical activity was required (e.g., pushing, pulling, turning, controlling, activating, etc.)? Was the task easy or demanding, slow or brisk, slack or strenuous, restful or laborious?</td>
</tr>
<tr>
<td>TEMPORAL DEMAND</td>
<td>Low/High</td>
<td>How much time pressure did you feel due to the rate or pace at which the tasks or task elements occurred? Was the pace slow and leisurely or rapid and frantic?</td>
</tr>
<tr>
<td>EFFORT</td>
<td>Low/High</td>
<td>How hard did you have to work (mentally and physically) to accomplish your level of performance?</td>
</tr>
<tr>
<td>PERFORMANCE</td>
<td>Good/Poor</td>
<td>How successful do you think you were in accomplishing the goals of the task set by the experimenter (or yourself)? How satisfied were you with your performance in accomplishing these goals?</td>
</tr>
<tr>
<td>FRUSTRATION LEVEL</td>
<td>Low/High</td>
<td>How insecure, discouraged, irritated, stressed and annoyed versus secure, gratified, content, relaxed and complacent did you feel during the task?</td>
</tr>
</tbody>
</table>
**Appendix D: T-Test Values**

### Independent Samples Test

<table>
<thead>
<tr>
<th></th>
<th>Levene’s Test for Equality of Variances</th>
<th>t-test for Equality of Means</th>
<th>95% Confidence Interval of the Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F</td>
<td>Sig</td>
<td>t</td>
</tr>
<tr>
<td>Depth</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Equal variances assumed</td>
<td>.116</td>
<td>.740</td>
<td>2.569</td>
</tr>
<tr>
<td>Equal variances not assumed</td>
<td>2.569</td>
<td>9.604</td>
<td>.029</td>
</tr>
<tr>
<td>Explanation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Equal variances assumed</td>
<td>3.125</td>
<td>.108</td>
<td>.166</td>
</tr>
<tr>
<td>Equal variances not assumed</td>
<td>.166</td>
<td>7.436</td>
<td>.872</td>
</tr>
<tr>
<td>Understandability</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Equal variances assumed</td>
<td>4.900</td>
<td>.051</td>
<td>.506</td>
</tr>
<tr>
<td>Equal variances not assumed</td>
<td>.506</td>
<td>7.364</td>
<td>.528</td>
</tr>
<tr>
<td>Reasoning</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Equal variances assumed</td>
<td>1.676</td>
<td>.224</td>
<td>3.969</td>
</tr>
<tr>
<td>Equal variances not assumed</td>
<td>3.969</td>
<td>6.516</td>
<td>.006</td>
</tr>
<tr>
<td>Process Order</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Equal variances assumed</td>
<td>1.364</td>
<td>.270</td>
<td>-1.969</td>
</tr>
<tr>
<td>Equal variances not assumed</td>
<td>-1.969</td>
<td>9.381</td>
<td>.079</td>
</tr>
</tbody>
</table>
Appendix E: VSCD User Manual

Introduction
The Verbal Source Code Descriptor (VSCD) is a tool used to help programmers verbally document software. This document provides guidance of how to work with the system.

Languages
The VSCD currently works with any programming language and there is not any specific compiler associated with this tool. Our goal in this stage is to evaluate the usability of the voice system in the programming environment. However in future versions of this system we will add compilers for specific programming languages to use with this tool.

Using the Verbal Source Code Descriptor

Starting
The system is accessed by running VSCD from the start menu. Once selected the main screen appears.

Closing
Clicking the Exit button in the toolbar or selecting the Project/ Exit menu from the menu bar can close the system. Do not close the system by clicking on the close (X) button on the upper right corner. If you do, so you will loose unsaved information.
Screen layout / description

When the system is run, the main screen appears.

The main screen has 5 areas.

Program Window

This window appears in the left side of the screen when a file is added to the project. The user types the body of the source code for the selected file in this window. When the user selects another file the system prompts the user with a message to save modifications before switching. The contents of this window are updated according to the contents of the selected file.

Description Window

This window appears on the right side of the Program Window. When the user adds a new file to the project the Program Window and Description Window open. The dictated text appears in the Description Window as the user types the source code in the Program Window. There is a description file associated with each program file, which contains the explanations and comments that the user dictates at the time of programming. When the user saves a file both the program file and description file are saved. There is a one by one relationship between each pair of program file and description file, which is saved in the system’s database.
Therefore, whenever you open a program file the system refers to the database to find the associated description file and loads it into the Description Window. Both the Program Window and Description Window can be edited and formatted using the edit and format tool in the Toolbar and menu bar.

Project Window

The Project Window displays a hierarchical list of the projects and all of the items contained in a project. There are two levels of hierarchy in the Project Window. The first level is the list of the project’s features, which specify the project’s behavior and capabilities. The second level is the list of files under each feature, which specifies the program functional components containing the source code and documentation.

Menu-bar

A menu-bar appears at the top of the form, which includes the following items:

**Project**

This menu provides the functions for manipulating projects. Each program is organized as of a project that consists of features and files. To start a program you should either create a new project or open an existing project.

![Project menu](Figure 2: Project menu)

**New Project**: Creates a new project
**Open Project**: Opens a dialog box to select an existing project to open.
**Add Feature**: Adds a new feature to the project. Each project is classified into features that represent the system’s externally visible behavior. For example, a program that reads employees’ information, calculates their salary and saves the information into database exposes the following features: read data, calculate salary, save data.
**Remove Feature:** Removes the selected feature from the list of features in the Project Window.

**Add File:** Adds a file under a specified feature. Before adding a file, select the appropriate feature from the Project Window.

**Save File:** Saves the file’s name and path into the system. If the file is being saved for the first time a dialog box will appear to specify the path and the name of the file.

**Save File As:** Saves a file in another path or name.

**Remove File:** Removes the selected file from the Project Window. However, the file is not deleted from its physical location if already saved.

**Save Project:** Saves the project information into a project file. A project file in this system is defined by .prj extension.

**Save Project As:** Saves the project in a new path or name.

**Close Project:** Closes the project and if its modifications are not saved, prompt the user with a message to save the modifications.

**Exit:** Closes the application.

**Edit**

This menu provides the user with common editing tools, as described below.

**Figure 3:** Edit Menu

**Cut:** Cuts the selected text into the system’s clipboard.

**Copy:** Copies the selected text into the system’s clipboard.

**Paste:** Pastes the contents of the system’s clipboard to the insertion point
Select All: Selects all the text in the window.

Find: Opens the Find form and asks for the text to be found and if it is replaced, asks for the text to be replaced.

![Find Dialog box](image)

**Figure 4: Find Dialog box**

Find Next: Finds the next occurrence of searched-for text.

Go to Row: Asks for the line number and moves the cursor to the appropriate line.

![Go to Dialog box](image)

**Figure 5: Go to Dialog box**

Format

This menu contains the functions and tools to format a document, as described below.

![Format Menu](image)

**Figure 6: Format Menu**
Verbal Source Code Descriptor

Align: Adjusts the alignment of the selected text to the Center, Left or Right of the screen.

Font: Opens a dialog box, as shown below, to set the font name, size and color of the selected text.

![Font Dialog box](image)

Figure 7: Font Dialog box

Background: Opens a dialog box, as below, to set the background color of the window.

![Background Dialog box](image)

Figure 8: Background Dialog box

Dictation

This menu sets the status of the voice system.

![Dictation Menu](image)

Figure 9: Dictation Menu
Verbal Source Code Descriptor

**Listening for dictation:** Activates the voice systems, and gets ready for the user to dictate.

**Not listening for dictation:** Stops dictation and deactivates the voice system.

You should ensure that listening for dictation is checked when using the ASR system.

**Tools**

This menu lets the user set up the system’s options, which at the moment, are only User options.

![Figure 10: Tools menu](image)

**User Options**

This tool helps to set up the current user from the list of users that has been already introduced to the ViaVoice engine. These users are defined in the user wizard of the ViaVoice application before the voice recognition training stage. It is necessary to set up the current user before working with the system to make sure the system is working with the right voice model. When you select the User Options menu, a dialog box appears, as shown in Figure 31 and you can select the current user from the list and press OK.

![Figure 11: Set Up User Dialog Box](image)

**Help**

This menu is unavailable at this stage. To get any help refer to the User Guide or ask the examiner.
**Tool Bar**

The toolbar provides similar tools as the menu-bar except for a key few tools. Figure 32 provides a labeled illustration of the tool bar. The labels in black are not available through the menu system because the user must select the sections to link and then select one of these tools. Providing these tools through the toolbar will speed up and facilitate these tasks.

![Tool Bar](image)

**Figure 12: Tool Bar**

**Link Source Code to Document**

This tool creates hyperlinks from source code to the documentation. This type of link can be used when a piece of code is explained or mentioned in a piece of comment or documentation. To make this type of link:

Select the appropriate piece of code or insertion point in the program window.
Select the appropriate piece of document to be linked and press the Link Source Code to Document button. A hyperlink indicator will appear in the insertion point or on top of the selected text in the program window. To check if your link was successful, click on the hyperlink indicator to bring up the appropriate piece of document which is linked to this point in the Description window.

**Link Document to Source Code**

This tool works the same way as the “Link Source Code to Document” but in a reverse order. It creates hyperlinks from the Comment/Document window to the program window but the process of linking is the same as described above. The hyperlink indicator for this type of link is. You usually create this type of link when a program variable or function name is mentioned in a piece of documentation and you would like to find out about the definition of the variable or function.

**Voice Commands**

The system is capable of recognizing some verbal commands as described below:

**Go to Sleep**
Verbal Source Code Descriptor

Prevents the voice system of dictation but the voice system remains active and waits for the “Wake up” command.

**Wake up**
Allows the voice system to dictate after it has been prevented by the “Go to Sleep” command.

**Select this**
Selects the word under or in front of the cursor.

**Go to Top**
Moves the cursor to the top of the document (Program or Descriptions).

**Go to Bottom**
Moves the cursor to the bottom of the document (Program or Descriptions).

**Cut this**
Cuts the selected text.

**Copy this**
Copies the selected text.

**Paste this**
Paste the clipboard contents in the insertion point.

**Delete this**
Deletes the selected text.
**Recording the Voice**

As you dictate to the voice recognition system you can record your voice using recplay.exe application. This is an application in the same path as VSCD and should be run before activating the voice recognition system. Figure 13 shows the form loaded after running recplay.exe.

![Record/Play Form](image)

**Figure 13: Record/Play Form**

When the form is loaded, you should specify the name and path of the audio file to be created in 'File to write audio to' text box; then press the start button. Next, minimize this window and continue working with VSCD. Recording the voice using recplay.exe does not interfere with simultaneous dictation to the voice recognition system. You should stop recording after deactivating the voice recognition system. The audio file is generated in the path specified by you in the form shown above.
Appendix F: Review of the features of some program understanding tools

This section summarizes the features and functionality of some source code understanding software tools currently available in academia and industry. I examine the capabilities of nine commercially available program understanding tools through their trial versions or documentation and present my findings in this report. These are the tools:

- CodeSurfer
- CC-Rider
- Exuberant Ctags
- Imagix 4D
- Java2HTML
- HyperCode
- IMPACT
- VBSurfer
- Project Analyzer

2.2.3.5.1. CodeSurfer

This is a C source code navigation and understanding tool developed by Gramma Tech Inc. (http://www.grammatech.com). It has a dependence analyzer that lets programmers analyze dependency among the various parts of the source code. It uses pointer analysis, web link navigation of code and integration with Emacs. It has also a scripting language for customization.

It provides a feature called "queries" by which you can find out what statements would influence or be influenced by a specific program point, immediately or transitively. Also one can see whether two points of a program would be influenced by each other immediately or transitively.

Everything is based on a project. The Project viewer allows users to browse a program's top-level structure and navigate to its function definitions. File viewer expands a file and shows the source code inside it.

CodeSurfer also provides users with navigation features to surf the results of each query, but not directly from the source code because there are so many of these links and displaying them within the code would destroy its readability. Instead, links are displayed in property sheets and pop-up menus. Blue items in property sheets are hot, and double clicking on them navigates to the corresponding text in the source code. Linked mode couples a property sheet updating with source code navigation. It actually helps to navigate search results by means of a linking strategy, not directly from the source code but from the property sheet.

This tool also has features, such as Finder, for collecting information about variables. This is an easy way to find out about variables from different perspectives.
Verbal Source Code Descriptor

For example, to get the declaration of a variable, type the variable name, or press the browse button in the Finder dialog box and check “Decls” in the setting options. A list of all declarations found for a given variable is created. Double clicking on a variable takes a user to the appropriate point of the source code.

Other settings are useful for obtaining more information about variables. These settings include “Uses”, which specifies direct or indirect uses of a variable. “Def” or “Kill” are program points where the value contained in a variable is necessarily changed. “Conditional kill” is a program point where the value contained in a variable may be changed. “Occurrence” is a program point where the name of the variable appears textually in the program. “Point to/Pointed” allows users to search for a variable. If variable P contains the address of variable X we say that P may point to X and X may be pointed to by P.

Note that items in the Finder result list are not hot, but can be navigated using the Next and Previous commands of Finder.

A property sheet is another facility for retrieving information about a variable. It lists all occurrences, uses, and definitions of a variable. Items in this sheet are hot and link to the corresponding program point by double clicking.

Additional variable usage information is available function-by-function in the project viewer. For example, the non-local variables that each function accesses or modifies are listed under its Dependence Point entry. Incoming variables lists the variables whose values may be used (immediately in the body of the function or transitively in a function called by the given function) when the function is called. The outgoing variable lists the variables whose values may be modified when the function is called.

Using CodeSurfer’s Dependence analysis feature users can specify how a part of a program or a variable can affect another parts of the program, or which parts will be affected by a variable or program point.

In addition to hyper link facilities provided indirectly in CodeSurfer, a global language-based context search feature explores the source code.

CodeSurfer works like a compiler. It parses and analyses all the source-code components of the system. The dependencies between statements in the program are computed by analyzing direct dependencies, then after carrying out a pointer analysis, by analyzing indirect dependencies. The results are stored in a data structure called “System Dependence Graph (SDG)”. The elements of the SDG are declarations, statements and conditions. These are linked to describe their data-dependence and control-dependence relationships. Sophisticated algorithms are used to traverse the SDG to answer queries. The result of a query evaluation is a set of SDG elements that can then be mapped back to locations in the source code.

CodeSurfer has a number of limitations mentioned in the literature. One is language because it only supports ANSI C. It does not yet support GNU C extensions, C++, or assembler code.

This product has several missing features.
There is no navigation between definitions and uses of types, and no navigation
between definitions and uses of macros.

There is no support for multi-user access to a project, yet there is no lock to detect
or prevent it.

The product is inaccurate in analyzing dependency because it uses a static analysis
that can miss some dependencies that arise at run-time.

2.2.3.5.2. CC-Rider

This is a C/C++ program visualization and source code comprehension software
tool developed by Western Wares (ccrider@westernwares.com).

This product contains the following features:

- Provides a wide variety of views, such as class hierarchy, class ancestry,
class nesting, name space topography, and call/caller trees. These views
include file relationships, program statistics, symbol definitions, symbol
references, template expansions, inherited class members and more.

- The documentation generation feature generates documentation
automatically in many formats (HTML, RTF, Windows Help format,
ASCII). This feature can also produce wall size tree charts, export tree
charts into documentation, and uses comments from source code.

- CC-Rider fully parses C and C++ for complete accuracy, including macro
and template expansions, string literal, and comments. It can handle over
1,000,000 lines of code.

- Provides users with system statistics, such as the number of source
modules, header files, tutorial definitions, total unique symbols, total
functions, total typedefs, and total macros.

- The visualizer feature provides graphical tree charts showing the
application’s structure. For example, the class hierarchy, which is a
graphical view of the class inheritance of a program, with the derived
classes are shown in a separate window (Multi-view feature). This
provides a useful overall view of the object-oriented structure of a C++
application.

- A symbol display window shows the actual definition of the class, as the
compiler views it, with any macro and template expanded. Double clicking
on a node in the class hierarchy tree brings it up. The description tab
shows any comments associated with a declaration. The reference tab
shows all the places where this class is referenced. It also shows the type
of each reference and where a variable is modified, expanded or read.

- Depending on the symbol type, other views for the symbol, such as the
class ancestry tree (e.g., the inverse of class hierarchy) can be accessed. It
is useful for exploring multiple inheritance designs.

- Nesting tree shows classes defined within the scope of other classes.
• Name space nesting tree provides a map to the name spaces and shows in which file a given symbol is located.

• CC-Rider shows how functions and variables are used in your application. Call/Caller tree shows data usage and how functions are called, providing a map to the procedural structure of an application. They diagram the logical flow of control, whereas File trees show physical file relationships.

• File Tree/File Used and Used by Trees show "#include" dependencies for one module. File Used By tree shows which file uses a particular header.

• Double clicking a file node brings up a concise summary of the file's contents. Various symbol types are defined, for example, all declarations/functions, type definitions, and classes in a specific file.

• Trees are helpful to give users a general overview of the application but this tool also provides users with detailed information about source code. In graph structure representation, we can double click on the name of any symbol and bring up the symbol display window. If we look at the list of references we can find all points where the specified symbol is called. By double clicking on the reference list we can see the actual source code where the function is called.

• CC-Rider examines every way that a function or data item is used in an application. It includes calls inside macro expansions and it avoids inactive references inside if/endif blocks.

• Comments are fully indexed allowing powerful documentation-based code browsing. For example, a Y2K problem is found by browsing comments and find related issues, and from there find the program point where the comment has been written, then find more information about the symbol that might seem dangerous and look at its references to see if there is any part in the application that refers to that symbol.

• String literal search helps to find a specific string literal using View/Browse String. This can be useful, for example, to determine where specific error messages are in an application.

• The documentation generation feature of this tool automatically generates program documents in several formats: Text, RTF, HTML and Win Help.

• Comments in the source code can be used to document each symbol. You determine what symbol you want to document, what elements to include in your manual, and how it is to be organized using the Documentation Options Dialog. The formatting of the document is controlled by a set of template files. These can be customized for your desired style, fonts and page layout.

• The other features of CC-Rider are automatic function prototyping, Symbol database API library, Command line utilities, and dBase export.
This tool is also language specific and works with the Standard C/C++ compiler. Compiler extensions are provided for Visual C++, C++ Builder, Watcom, IBM, Franklin, Microware, GNU, and Metaware. The interface of this tool is confusing in that it is difficult to find all features of this tool quickly. It uses hypertext technology but only implicitly. There is no linking information in the source code.

2.2.3.5.3. Exuberant Ctags

(Thectags.html)
The Ctags and etags programs generate an index or “tag” file for a variety of language objects found in file(s). This tag file allows the objects to be quickly and easily located by a text editor or other utility. A “tag” signifies a language object for which an index entry is available. Alternatively, Ctags can generate a cross-reference file, which lists information in readable form about the various source objects found in a set of language files. Tag index files are supported by numerous editors, which allow the user to locate the object associated with a name appearing in a source file and go to the file and line which defines the name. The editors supported in the most recent release of the software include Vi and its derivations (Elvis, Vim, Vile, Lemmy), CRISP, Emacs, FTE (Folding Text Editor), JED, and NEdit. Ctags is capable of generating different kinds of tags for each of many different languages.

As Ctags considers each file name in turn, it tries to determine the language of the file by applying the following three tests in order:

1) If the file extension has been mapped to a language,
2) If the file name matches a shell pattern mapped to a language, and finally if the file is executable and its first line specifies an interpreter using the Unix-style "#!/" specification (if supported on the platform).
3) If a language was identified, the file is opened and then the appropriate language parser is called to operate on the currently open file.

The parser parses through the file and adds an entry to the tag file for each language object it is written to handle. The TAG FILE FORMAT then can be read by one of the specified editors and can be manipulated using the editor commands.

This technique is very similar to HTML, however, it is limited to those specific editors mentioned above. These editors need specific commands to use and navigate a tag file and do not present a visual display, such as hypertext links appearing in an HTML document in a browser, nor do they provide color-coding that makes a source code more readable.

2.2.3.5.4. Imagix 4D

Imagix 4D is a tool for C/C++ languages produced by Imagix Corporation (www.imagix.com). The Imagix 4D interface contains display windows that are divided to two parts:
Verbal Source Code Descriptor

1) A graph window that shows a group of symbols (function, variables, and classes) and how they relate to each other. Symbols are shown in different colors and shapes.

2) The list window complements the graph window by showing the context of the symbols displayed in the graph window. Users can see where each symbol is located in the physical file structure and the members that make up specific files, classes or structures.

The tool includes two browsers, one for classes and one for files. The browser provides information about files and classes using three different panels representing three different views. The Index panel is a list of all files currently loaded into the Imagix 4D’s database. The Relationship panel shows how one particular file or class is related to others (#Include files inside a file). The Members panel enumerates all symbols defined or declared in that file or class.

When using a normal editor with Imagix 4D, its built-in File Editor facility enables users to achieve understanding more rapidly as source code is examined and modified. Color-coding makes it easy to distinguish between comments and actual code. Symbols (classes, functions, variables, types) are also coded in different colors.

Another way to view the source code is through the Use of Browser. It shows where a symbol is declared, its definition, and who calls it. There are other uses of symbols such as Set, Read, Used, Friend, and etc.

Imagix 4D’s Flow Chart shows the internal flow of control within functions. This could be useful to explore the logic flow of a complex function. It also provides a Comprehensive report about software such as a list of functions in its database, the number of lines each has, and how many callers they have. Imagix 4D has an entity/relationship/attribute database that can be used to represent software from different perspectives. The database is built by the Imagix 4D parser, which works much like a compiler.

The Graph window as discussed previously shows symbols and their relationships. The Graph Symbols Key is a useful tool for tracking the symbols displayed in the Graph window. It shows symbol properties and also the relationships between them. A specific color and shape is allocated to each symbol consistent over the entire system. For example, blue cubes are used for functions. Different colors are also used to indicate the type of relationship between two symbols. For example, the Red arrow shows function calls.

Users can indicate the type of symbols that are visible in the Graph window to graphically examine different aspects of your software.

The list window describes various characteristics about the symbols themselves. The Imagix 4D’s database tracks the location of each symbol. For example, variable A is defined in Class B. Class B is declared in file C, located in directory D. List window allows users to examine this hierarchy of containers.
The Imagix 4D’s database also includes information about the specific attributes of each symbol. For example, it tracks how many lines of code a function has, the scope of a variable and the date a file was last modified. This information can be viewed in a List window.

The Browser (File & Class) provides information about files and classes. It has 3 panels: Index, Relationships and Members.

There is a File Editor that shows the actual source code, which is color-coded to increase readability. This editor also supports hypertext source code navigation. By double clicking on a colored symbol, the File Editor immediately browses to the file and line number where that symbol is defined. This navigation is possible from any of the other Imagix 4D displays (the Graph window, the List window, the Browser, the Flow Chart and reports).

Another form of navigation uses Next and Prev buttons on the icon bar of the File editor. These buttons enable users to cycle through all locations where a symbol appears. The Imagix 4D database contains information about all locations where a symbol is defined, declared, called, read, written, or used as a type. This tool also has a simple grep search that finds all literal matches for a string. It misses any matches hidden through macro definitions.

With Imagix 4D, users can have multiple (up to 10) File Editor windows open at a time. This enables simultaneous and concurrent viewing and manipulation of pieces of code in separate windows.

A Flow Chart tool that shows the control flow and logical structure of a program is provided with Imagix 4D. The Flow Chart window is linked to the File Editor. Clicking on a symbol in the flow chart moves the cursor in the File Editor to a location where the symbol is defined. Likewise clicking on the source code causes the matching symbol in the Flow Chart window to be highlighted.

Imagix 4D operates in 4 modes: Explore, Analysis, Control Flow, and Browse modes. Explore mode enables users to systematically study software on many levels. Analyze mode examines the quantitative characteristics of the software. Control Flow mode provides users with the sequences and conditions of function calls and variable usage. Browse mode helps users navigate through source code and find the key relationships between classes, functions, variables and types. In Browse mode all of Imagix 4D’s display windows are linked. For example, if a symbol is the focus point in Graph window, the File editor automatically displays the source code for the specified symbol (the symbol is highlighted in yellow wherever it appears in the display).

2.2.3.5.5. Java2HTML
(http://www.java2html.com)
This is a tool that converts Java source code into color-coded and browsable HTML representations. The major features of this tool are as below:

1. Converts Java code into color-coded HTML.

2. Specifies TAB spacing.
3. Specifies style and color options of all Java keyword types via a standard Cascading Style Sheet.
5. Links class references into source code for class definitions.
6. Links external Class references into "JavaDoc" output.
7. Optional line number feature.
8. Produces a "Stylesheet.css" in the output directory, which can be edited to change color or style scheme (This file will get overwritten if it already exists when you run j2h program).

The limitations and future works of this product have been reported as follows:
1. Support for Apple products and Java properties file to specify options.
2. Make use of a proper installer package.
3. Method and object instance hyper links.
4. Hyperlinks of references within comments and strings.
5. Style sheet option to prevent the overwriting of a customized Style sheet.
7. Extend Java2HTML to deal with other languages.
8. Possibly publish the source code with an appropriate license.
9. HTML driven search facility.
10. Allows an option to specify a file, which contains a list of directories, and/or files to be included in the Java2HTML process. This allows users to be exact about what they want processed.

2.2.3.5.6. HyperCode

HyperCode provides a hypertext representation of source code. It uses HTML technology so that standard browsers such as Internet Explorer can be used to display program artifacts and control flows. Supporting HyperCode with a code database front-ended by a WWW server enables software sharing and development on a global scale by leveraging the programming, debugging, and computing power brought together by the World Wide Web. HyperCode has been defined as “HTML-based hypertext representation of program source code richly decorated with informative links.” [58]

This technique is not only a powerful method to comprehend a program but has also become a tool for “code sharing and development”, due to the support of a WWW server.

The basic HyperCode system provides different types of links: Function links, Data type links, Variable links, and Macro links.
Function links connect function calls to corresponding definitions. This enables users to extract the entire function dependency graph of a program. These links are bi-directional, so that there are links from function definition to a "backtrack" list consisting of the sites from which the function is called. The user can click on any of the items in this list and jump to the appropriate location in the source code.

Data type links provided by HyperCode explicitly represent an important aspect of program structure; specifically data type links encode the entire set of data structure relationships used by a program.

These links are useful to make any modification in a particular structure and evaluate the effects of changes on every dependent data structure and variable. Variable links are the same as other links. The paper basically focuses on C programming language features and provides examples from C structures.

Macro links are the same as other types of links, but the system represents the macro expansion in multi levels.

For example, this tool can build the hypertext representation of a C program by using C pre-processor and compiler. This mechanism uses the information being gathered during pre-processing and compiling the source code. This information stores in auxiliary files, and before the linking stage (that makes an executable file of the source code), a component called the HyperCode generation program gets the auxiliary files and creates the hyper links.

Some C language specific features complicate the HyperCode generation process. For example, the #if and #ifdef pre-processor constructs can change the semantics of a piece of code at compile-time. This was considered an open problem in 1994, the time that this work is reported. In addition to the features of HyperCode for understanding and developing source code, it can be used as a framework for global code sharing, by replacing the mechanism of static HTML files with a database of code front-ended by a WWW server.

2.2.3.5.7. IMPACT

IMPACT is built by ADPAC Corporation (www.adpaccorp.com) and works on mainframe systems. When any data element requires change, it automates the process of locating every occurrence of that element and all of its aliases, throughout an entire application. Based on estimated parameters, IMPACT assesses the cost of projected changes and details the resources required to accomplish them. It offers "what if" modeling of resource and cost assumptions. It handles COBOL, PL/1, and assembler, and almost any other textual computer language.

IMPACT assesses the impact, automates change and more:

- Locates all data elements and their procedural uses throughout the application.
- Identifies each affected element's alternative alias names.
Verbal Source Code Descriptor

- Produces complete impact reports, including budget estimates.

2.2.3.5.8. VBSurfer

VBSurfer is a tool produced by Logiciels Ntech Software (http://www.vbsurfer.com/) for Visual Basic source code representation.

VBSurfer starts from a VB project and exposes the following features:

- Presenting a Hyperlinked Structure for the VB source files.
- Highlighting Comments, strings, VB reserved words, local variables and so on. Furthermore, VBSurfer takes care of structure indentations.
- Surfing your VB project.
- Representing children-parent relationships between subprogram and functions.
- Documentation and source code on the same page.
- Sharing your work with others.
- The capability to collapse/expand While, If and so on structures with a single click.

2.2.3.5.9. Project Analyzer

Project Analyzer is a tool to document and optimize visual basic code produced by Aivosto Oy (http://www.aivosto.com/vb.html).

With Project Analyzer's problem detection feature, you remove unnecessary code, get recommendations for better coding style, and check for error prone places in your project. It lets you browse through code as hypertext or a graphical call tree. For fine-tuning and monitoring software development, Project Analyzer provides industry standard quality control metrics. It also documents your projects with cross-references and other useful data.

The analysis consists of two phases:

1. Gathering basic data about subs, variables etc.
2. Finding cross-references.

Project Analyzer makes a full code review. It suggests and performs numerous code improvements, leading to faster and smaller programs and enforcing adherence to programming standards.

Project Analyzer generates technical documentation by reading program source code. The available documents include graphical representations of program structure, commented source code listings and various reports such as file dependencies.

Project Analyzer enables programmers to learn to understand existing code in less time. By browsing code in hypertext form, a programmer can quickly understand how a certain function operates with other functions and variables.
The following is the list of the features offered by Project Analyzer:

- **Problem detection:**
  
  Project Analyzer detects several types of programming problems. The problems are of four categories:
  
  1. **Optimization.** These problems affect the speed and size of the resulting program negatively.
  2. **Style.** These problems are related to the programming style. They do not necessarily lead to problems in the short run, but they often lead to lower understandability and errors.
  3. **Metrics.** Metrics is a sub-category of Style. Target values can be set for different metrics and be monitored if some part of your program exceeds the limits.
  4. **Functionality.** These problems affect the run-time functionality of the program.
  5. **.NET compatibility.** These problems show incompatibilities between VB6 and VB.NET syntax. Enterprise Edition is required to view these problems.

- **Metrics:** To monitor programming performance, programmers often use simple metrics, such as lines of code or EXE file size; however, this is not always enough. Project Analyzer helps to monitor the understandability, complexity and reusability of the source code.

- **Call trees:** to show dependencies, or cross-referencing, between procedures or files in the project as call trees.

- **Reports:** Project Analyzer can produce several kinds of reports. All reports are available in several report formats. These reports are accessible through the display, the printer, or a file for a word processor, WWW browser or Help compiler.

- **VB Browser:** Project Analyzer analyses a project and gives a list of all components (Forms, Modules, Classes etc.) of the project.

- **Archive project files**

- **Options:** You can set up your system’s features using the options provided in the menu bar. These Options are as follows:
  
  - **Report to**
    
    - Display
    - Printer
    - File
  - **Printer Set up**
  - **General Options**
- Font Options
- Hypertext Options
- HTML Options
- Problem Options

- Saving and reloading of analysis
- Module or procedure.
A number of code formatting and programming style guidelines have been proposed in past decades such as indentation, alignment, commenting, and control structure [15, 16, 17, 18]. However, many of these recommended techniques have not been empirically or theoretically verified. “Programming style rules are indefinite, inconsistent, and contradictory” [14, pg. 244]. For example, there are contradictory ideas about different styles such as indentation, alignment, commenting, and control structures [6, 15, 16, 17, 18]. If a programmer learns and uses one style, but a user has learned and uses a different and contradictory one, the user may be confused and misinterpret the intended purpose behind a specific style instance used by the original programmer. This may reduce, rather than increase, the readability of the program.

Oman and Cook [14, pg. 244] state, “Research in programming style needs to follow a common paradigm in order to reduce, if not eliminate, the disparity found in programming style research”. They carried out a semantic analysis of stylistic factors and determined how specific style factors affect comprehension. This analysis consists of five stages:

1. Identifying and categorizing the factors comprising programming style.
2. Determining the effects of these stylistic factors on program complexity and comprehension.
3. Establishing principles of good style and subsequent style standards (relative to local constraints) based on the results from 2.
4. Measuring the adequacy and completeness of the standards.
5. Measuring the degree of conformity between the code and the standards.

Oman and Cook [14] prepared a list of all programming styles from books and resources and then classified the style factors into style categories by considering their common principles. Their research resulted in style taxonomy containing four major categories:

- General programming practices (“Rules and guidelines pertaining to the programming process that directly affect the style of the product” [14, pg. 284]. For example “First understand the problem; don’t start coding right away” [14, pg. 249])

- Typographic style (“Style characteristics affecting only the typographic layout and commenting of code with no affect on program execution” [14, pg. 284]. For example “Do not put more than one control statement per line.” [14, pg. 249]

- Control structure style (Style characteristics pertaining to the choice and use of control flow constructs, the manner in which the program or system
is decomposed into algorithms, and the method in which those algorithms are implemented. This excludes data structure aspects.” [14, pg. 284]. For example “Use library functions”)

- Information structure style (“Style characteristics pertaining to the choice and use of data structure and data flow techniques.” Such as “Validate input for legality and plausibility.”[14, pg. 249])

These four categories are then divided further into micro/macro classes. Macro class guidelines pertain to the whole program or system, while the “micro” class guidelines pertain to those relevant to a single program module or statement.

In order to assess the completeness of the taxonomy, Oman and Cook gathered approximately 236 programming rules and categorized them using their pre-defined taxonomy. They also tracked the frequency of the appearance of each rule in the literature. This helped identify unresolved issues and conflicts in the published style rules, to design an empirical study on programming styles, and to provide a framework for teaching and using programming styles.

This taxonomy is only the first step of programming style research. The next step is to analyze the taxonomy and fill the gaps in our understanding of programming style. This analysis must consider program comprehension theories to find those rules and styles that are compatible with programmer comprehension theory, which could lead to more effective and consistent style rules and guidelines. Once the rules are developed, useful tools can be created to implement the rules and guidelines.
# Appendix H: Comment Generated by VSCD

## Group

### PROGRAM 1

<table>
<thead>
<tr>
<th>Source Code</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Public function Cos(byval X as double) as Double</code></td>
<td>OK I'm Going to Develop A Function to calculate Cos(X). I need 2 functions. One for power and another for factorials. Then I'll mix them in another function called Cos. The Main Program is a function that I call Cos in it. OK Now I want to develop the function Cos itself and Then I'll develop Some other functions. This is the header of my function. This is a variable which is X as a double Variable. It is my loop counter. I call it i. It's as an integer variable. I'm going to make the main loop. I will calculate the Cos using Power and Factorial functions. The Lowerbound is 1 and the upperbound is 10. I define a temporary variable to save the value of cos in each iteration of the loop. I examin the iteartion if its odd or even. I change the bopundry of the loop to 5. because I multiply X with the loop counter. I change the loop counter. I fix the loop counter to make calculation easier, lowerbound is 0 and upperbound is 10 and setp is 2. I return the value after loop. No it's time to develop Power nad factorial functions. first I develop power function. It has two parameters. One is the number itself and the other is Power. I define them as double. Again I define loop counter. I call it i and as an integer variable. And again we have a temporary variable here. I initialize the temporary variable. I start making the loop. Next step. I start making factorial function. This function has just one parameter. I call it Param1 as double. OK, Factorial is completed too. Right now we have three functions we are ready to make Main function. I develop the main function in this way: I make an array. I put the initial data in it and then I use it in a loop. It's kind of a sub routine. I call it Main. I define a variable. It's an array. It has 10 double numbers. It stores the result values.</td>
</tr>
<tr>
<td><code>Dim i as integer</code></td>
<td></td>
</tr>
<tr>
<td><code>Dim dblTemp as double</code></td>
<td></td>
</tr>
<tr>
<td><code>dblTemp=0</code></td>
<td></td>
</tr>
<tr>
<td><code>For i = 0 to 10 step 2</code></td>
<td></td>
</tr>
<tr>
<td><code>if (i mod 2)=0 then</code></td>
<td></td>
</tr>
<tr>
<td><code>dbltemp = dbltemp - (power(x,i)/factorial(i))</code></td>
<td></td>
</tr>
<tr>
<td><code>else</code></td>
<td></td>
</tr>
<tr>
<td><code>dbltemp = dbltemp + (power(x,i)/factorial(i))</code></td>
<td></td>
</tr>
<tr>
<td><code>end if</code></td>
<td></td>
</tr>
<tr>
<td><code>Next i</code></td>
<td></td>
</tr>
<tr>
<td><code>Cos=dblTemp</code></td>
<td></td>
</tr>
<tr>
<td><code>End Function</code></td>
<td></td>
</tr>
<tr>
<td><code>Public Function Power(byval dblam1 as Double, byval intParam2 as byte) as Double</code></td>
<td></td>
</tr>
<tr>
<td><code>Dim i as Integer</code></td>
<td></td>
</tr>
<tr>
<td><code>Dim dblTemp as Double</code></td>
<td></td>
</tr>
<tr>
<td><code>dblTemp=1</code></td>
<td></td>
</tr>
<tr>
<td><code>For i=1 to intParam2</code></td>
<td></td>
</tr>
<tr>
<td><code>dblTemp = dblTemp*dblParam1</code></td>
<td></td>
</tr>
<tr>
<td><code>Next i</code></td>
<td></td>
</tr>
<tr>
<td><code>Power = dblTemp</code></td>
<td></td>
</tr>
<tr>
<td><code>End Function</code></td>
<td></td>
</tr>
<tr>
<td><code>Public Function Factorial() (byvaldblParam1 as Double) as Double</code></td>
<td></td>
</tr>
<tr>
<td><code>Dim i as Integer</code></td>
<td></td>
</tr>
<tr>
<td><code>Dim dblTemp as Double</code></td>
<td></td>
</tr>
<tr>
<td><code>dblTemp=1</code></td>
<td></td>
</tr>
<tr>
<td><code>For i=1 to intParam1</code></td>
<td></td>
</tr>
<tr>
<td><code>dblTemp = dblTemp*i</code></td>
<td></td>
</tr>
<tr>
<td><code>Next i</code></td>
<td></td>
</tr>
<tr>
<td><code>Power = dblTemp</code></td>
<td></td>
</tr>
<tr>
<td><code>End Function</code></td>
<td></td>
</tr>
<tr>
<td><code>Sub Main()</code></td>
<td></td>
</tr>
<tr>
<td><code>Dim Data(10) as Double</code></td>
<td></td>
</tr>
</tbody>
</table>

---

97
Verbal Source Code Descriptor

```vbnet
Dim i as Integer
  Dim Result(10) as Double
  Data(1) = 0
  Data(2) = 3.14 / 6
  Data(3) = 3.14 / 5
  Data(4) = 3.14 / 4
  Data(5) = 3.14 / 3
  Data(6) = 3.14 / 2
  Data(7) = 3 * 3.14 / 4
  Data(8) = 3.14
  Data(9) = 3 * 3.14 / 2
  Data(10) = 2 * 3.14
  For i = 1 to 10
    Result(i) = Cos(Data(i))
  Next i
End Sub
```

```vbnet
Public function Cos (byVal X as double) as Double
  { "This is the header of my function. This is a variable which is X as a double Variable."
  Dim i as integer
    { "It is my loop counter. I call it i. It's as an integer variable."
    For i = 0 to 10 step 2
      if (i mod 2) = 0 then
        dbltemp = dbltemp - (power(x, i) / factorial(i))
      else
        dbltemp = dbltemp + (power(x, i) / factorial(i))
      end if
    Next i
    Cos = dblTemp
    { "I return the value after loop."
  }
  Public Function Power { "I need 2 functions. One for power and another for factorials."
    ...
  }
  ...
  Public Function Factorial { "...
    ...
  }
  Dim i as Integer
    { "Again I define loop counter. I call it i and as an integer variable."
  Dim dblTemp as Double
    { "And again we have a temporary variable here."
  dblTemp = 1
    { "I initialize the temporary variable."}
```
Verbal Source Code Descriptor

For $i=1$ to intParam2
    dblTemp = dblTemp * dblParam1
Next i

I start making the loop.

Sub Main()
Main Program is a function that I call Cos in it.

Dim Result(10) as Double
I define a variable. It's an array. It has 10 double numbers. It stores the result values.

PROGRAM 2

<table>
<thead>
<tr>
<th>Source Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class AngleCOSCalculator {</td>
<td>I need to write a class that has three main methods; one for calculating factorials, one for calculating powers, and another for generating the cosine of the angle.</td>
</tr>
<tr>
<td>public double factorial(double x) {</td>
<td>The factorial method in takes a single argument: the integer with which to calculate. The for loop needs to iterate through the code X number of times. The method needs to return the results as an integer.</td>
</tr>
<tr>
<td>double result;</td>
<td>The power method will return an integer and take the base and exponent as arguments.</td>
</tr>
<tr>
<td>for (int it = 1; it &lt;= x; x++) {</td>
<td>The cosine method will return a double array and take an array of integers as an argument. I also need a method that will calculate the sign of the angle.</td>
</tr>
<tr>
<td>result = result * it;</td>
<td>If the remainder is zero, return a positive value, otherwise return the negative.</td>
</tr>
<tr>
<td>}</td>
<td>I'll need a main entry point into the program that will put together the array of angles to pass to the cosine method. I'll declare *pi as 3.14, which is good enough. So I need to calculate each of the angles by hand.</td>
</tr>
<tr>
<td>return result;</td>
<td>The first one is zero, the next one is pi divided by six, and so on until pi divided by two.</td>
</tr>
<tr>
<td>}</td>
<td>I think I'll refactor that into a for loop and save myself some typing. Icremement the angle number, do the division...</td>
</tr>
<tr>
<td>public int power(int base, int exponent) {</td>
<td>So the for loop was too much work, so I just calculated</td>
</tr>
<tr>
<td>int result;</td>
<td></td>
</tr>
<tr>
<td>for (int it = 1; it &lt;= exponent; it++) {</td>
<td></td>
</tr>
<tr>
<td>result = result * base;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>return result;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>public int sign(int x) {</td>
<td></td>
</tr>
<tr>
<td>if ((x % 2) == 0)</td>
<td></td>
</tr>
<tr>
<td>return 1;</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>return -1;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>public double[] angleCOS(double[] angles) {</td>
<td></td>
</tr>
<tr>
<td>int size = angles.size;</td>
<td></td>
</tr>
<tr>
<td>double[] cosigns;</td>
<td></td>
</tr>
<tr>
<td>for (int it = 0; it &lt; 10; it++) {</td>
<td></td>
</tr>
<tr>
<td>double val = angles[it];</td>
<td></td>
</tr>
<tr>
<td>int result;</td>
<td></td>
</tr>
<tr>
<td>for (int x = 0; x &lt; 5; x++) {</td>
<td></td>
</tr>
<tr>
<td>result = (sign(x) * power(angle, 1<em>x/2) / factorial(1</em>x/2));</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>cosigns[it] = result;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>return cosigns;</td>
<td></td>
</tr>
</tbody>
</table>
```java
public static void main(String[] args) {
    double pi = 3.14;
    double[] angles =
        {0, pi / 6, pi / 5, pi / 4, pi / 3, pi / 2, 3 * pi / 4, pi, 3 * pi / 2, 2 * pi};
    double[] cosigns = this.angleCOS(angles);
}
```

I need a for loop that will iterate through each element in the angles array and do the calculation. Get the value out of the array and calculate based on the formula given. Calculate the result by calling the sign method multiplied by the power of the angle divided by the factorial and package the results back up into the array.

So I’ll need to refactor the factorial method to take a double instead of an int. The factorial method also needs to return a double.

That’s it.

---

I need to write a class that has three main methods; one for calculating factorials, one for calculating powers, and another for actually generating the cosine of the angle.

```java
public double factorial(double x) {
    int it = 1;
    result = result * it;
}
```

The factorial method in takes a single argument: the integer with which to calculate.

```java
public double factorial(double x) {
    int it = 1;
    result = result * it;
}
```

So I’ll need to refactor the factorial method to take a double instead of an int. The factorial method also needs to return a double...

```java
for (int it = 1; it <= x; x++) {
    result = result * it;
}
```

The for loop needs to iterate through the code X number of times. The method needs to return the results as an integer.

```java
public int power(int base, int exponent) {
    int it = 1;
    result = result * it;
}
```

The power method will return an integer and take the base and exponent as arguments.

```java
public int sign(int x) {
    int it = 1;
    result = result * it;
}
```

I also need a method that will calculate the sign of the angle.
if ((x % 2) == 0)
    return 1;
else
    return -1;

public double[] angleCOS(double[] angles)
{
    ...
}

if the remainder is zero, return a positive value, otherwise return the negative.

The cosine method will return a double array and take an array of integers as an argument.

I need a for loop that will iterate through each element in the angles array and do the calculation.

Calculate the result by calling the sign method multiplied by the power of the angle divided by the factorial and package the results back up into the array.

I need to store the value of each of the angles. The first one is zero, the next one is pi divided by six and so on. I think I'll put this into a for loop and save myself some typing. Increment the angle number and do the division. No the for loop was too much work, so I just calculated them one by one...

PROGRAM 3

<table>
<thead>
<tr>
<th>Source Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>define TRUE 1</td>
<td></td>
</tr>
<tr>
<td>#define FALSE 0</td>
<td></td>
</tr>
<tr>
<td>#define PI 3.1416</td>
<td></td>
</tr>
</tbody>
</table>
| int main()
{
    double values[0, PI/6, PI/5, PI/4, PI/3, PI/2, 3*PI / 4, |
| First I'm going to write a function to calculate the powers. Actually no, first I'm going to write the function that does the factorials. So int factorial and... I'll name this number number num. So for i equal to num... wait... int i. Ok, so i is going to be the iterator. So, and num is going to be the factorial. So for i equals num, until i is equal 0, no while, while i is bigger than 0 and you keep doing it and you do i minus minus. And then... We need a |
Verbal Source Code Descriptor

```c
int factorial(int num)
{
    int i = 0; total = 1;
    for (i = num - 1; i > 0; i--)
        total *= i;
    return total;
}

double power(double x, int pow)
{
    int i = 0, total = x;
    for (i = 0; i < pow; i++)
        total *= x;
    return total;
}

double findCos(double value)
{
    int i = 0, add = FALSE;
    double total = 1;
    for (i = 2; i <= 10; i += 2)
    {
        if (add)
            total += power(value, i) / factorial(i);
        else
            total -= power(value, i) / factorial(i);
        add = !add;
    }
    return total;
}

int main()
{
    double pi = 3.1416;
    double value[] = {pi / 2, 2 * pi};
    int i, elements = 10;
    for (i = 0; i < elements; i++)
        printf("%d\n", findCos(value[i]));
    return 0;
}
```

The code calculates the factorial of a number using a loop, the power of a number using another loop, and the cosine of an angle using a loop and a recursive function. The main program iterates through a set of values, printing the cosine of each angle.
elements. And now for each of them, find it's cos. So to
do that, you just go 14 printf a double and the cos of the
value at i0. Done.

int i, elements

printf("%d
", findCos(value[i]));

int i, Total;

for (i=num - 1; i>0; i--)
  0 total = total * i;

return total;

double power(double x, int pow)
{
  ...
}

int i,

total = total * x;

puble findCos(double value)

add = FALSE;

double power(double x, int pow)
  = Next I'll write the function that
calculates the powers.

for (i=0; i<pow; i++)
  = So I'll put it in a loop

return total;

int i, total;

= first I'll declare another total and
iterator.

total = total + power(value, i) / factorial(i);

= then you add it together so you go
total is equal to total plus the power of
the value.

#define PI 3.1416
= define pi as well 3.1416.

---

**PROGRAM 4**

<table>
<thead>
<tr>
<th>Source Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>import java.Math.*;</td>
<td>the purpose of this program is to generate the</td>
</tr>
</tbody>
</table>
import java.lang.*;

public class TestCOS {

public static void main (String args []) {

    double arrValues [] = {0,pi/6,pi/5,pi/4,pi/3,pi/2,3pi/4,pi,3pi/2,2pi};

    for (int i=0;i<arrValues.length;i++){
        System.out.println("The COS of "+arrValues[i] +" is: "+
calcCOS(arrValues[i]));
    }
}

    public double calcCOS (double x){
        return (1-(x^2/2!)+(x^4/4!)-(x^6/6!)+(x^8/8!)-(x^102/10!));
    }
}

The purpose of this program is to generate the cosine of 10 ten numbers by calling a cosine subroutine.

import java.Math;
import java.lang;

public double calcCOS (double x){
    return (1-(x^2/2!)+(x^4/4!)-(x^6/6!)+(x^8/8!)-(x^102/10!));
}

calcCos is the subroutine which calculated the cos of an angle, it uses a power divided by factorial algorithm. It takes in a double and return to double as well.

Double x will be the the value returned taken in calcCos will be called in the main program. The main contains an array of values called arrValues which will be used for calcCos.

Now we have a print statement, which is gonna tell us what the initial angle is and the cos of it. This print statement is going to be in a for loop which will iterate the length of the array. So within the print statement we have a call to calcCos and this will calculate the cosine of each value in the array.
Verbal Source Code Descriptor

<table>
<thead>
<tr>
<th>PROGRAM 5</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Source Code</strong></td>
<td><strong>Definition and initialize the array for all the angles of x (10 values)</strong></td>
</tr>
<tr>
<td>dim angle(10) as double</td>
<td>The i loop calculates all n values for each of the 10 angles</td>
</tr>
<tr>
<td>dim pi as double</td>
<td>The n loop calculates all parts of the cos equation for each i angle</td>
</tr>
<tr>
<td>dim cosofx(10) as double</td>
<td>Values for n are 2, 4, 6, 8, and 10 (skips values by 2)</td>
</tr>
<tr>
<td>dim factorial as double</td>
<td>Each n value (2, 4, 6, 8, 10) has different factorial values so calculate them individually</td>
</tr>
<tr>
<td>dim temp(5) as double</td>
<td>If n is 4 or 8 then have positive fractions, if n is 6 or 10 then have negative fractions</td>
</tr>
<tr>
<td>dim i, n as integer</td>
<td>Calculate the temporary cosofx per value of angle through using temp(n)</td>
</tr>
<tr>
<td>pi = 3.14</td>
<td><strong>for i = 0 to 9</strong></td>
</tr>
<tr>
<td>⇒ angle(0)=0</td>
<td><strong>for n = 2 to 10 step 2</strong></td>
</tr>
<tr>
<td>angle(1)=pi/6</td>
<td>if n = 2 then</td>
</tr>
<tr>
<td>angle(2)=pi/5</td>
<td>factorial = 2</td>
</tr>
<tr>
<td>angle(3)=pi/4</td>
<td>⇒ temp(n) = 1 - angle(i)^n / factorial</td>
</tr>
<tr>
<td>angle(4)=pi/3</td>
<td>if n = 4 then</td>
</tr>
<tr>
<td>angle(5)=pi/2</td>
<td>factorial = 4 * 3 * factorial</td>
</tr>
<tr>
<td>angle(6)=3*pi/4</td>
<td>⇒ temp(n) = angle(i)^n / factorial</td>
</tr>
<tr>
<td>angle(7)=pi</td>
<td>if n = 6 then</td>
</tr>
<tr>
<td>angle(8)=5*pi/2</td>
<td>factorial = 6 * 5 * factorial</td>
</tr>
<tr>
<td>angle(9)=2*pi</td>
<td>⇒ temp(n) = angle(i)^n / factorial</td>
</tr>
<tr>
<td>  for i = 0 to 9</td>
<td>if n = 8 then</td>
</tr>
<tr>
<td>  for n = 2 to 10 step 2</td>
<td>factorial = 8 * 7 * factorial</td>
</tr>
<tr>
<td>if n = 2 then</td>
<td>⇒ temp(n) = angle(i)^n / factorial</td>
</tr>
<tr>
<td>factorial = 2</td>
<td>if n = 10 then</td>
</tr>
<tr>
<td>⇒ temp(n) = 1 - angle(i)^n / factorial</td>
<td>factorial = 10 * 9 * factorial</td>
</tr>
<tr>
<td>factorial = 4 * 3 * factorial</td>
<td>⇒ temp(n) = -1 * angle(i)^n / factorial</td>
</tr>
<tr>
<td>factorial = 6 * 5 * factorial</td>
<td>cosofx(i) = cosofx(i) + temp(n)</td>
</tr>
<tr>
<td>factorial = 8 * 7 * factorial</td>
<td>next n</td>
</tr>
<tr>
<td>factorial = 10 * 9 * factorial</td>
<td>picture1.print &quot;The cos of angle &quot; &amp; angle(i) &amp; &quot; is &quot; &amp; cosofx(i) &amp; vbcrlf</td>
</tr>
<tr>
<td>Dim i, n as integer</td>
<td>next i</td>
</tr>
<tr>
<td>Values for n are 2, 4, 6, 8, and 10 (skips values by 2)</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>angle(0)=0</code>&lt;br&gt;<code>angle(1)=pi/6</code>&lt;br&gt;<code>angle(2)=pi/5</code>&lt;br&gt;<code>angle(3)=pi/4</code>&lt;br&gt;<code>angle(4)=pi/3</code>&lt;br&gt;<code>angle(5)=pi/2</code>&lt;br&gt;<code>angle(6)=3*pi/4</code>&lt;br&gt;<code>angle(7)=pi</code>&lt;br&gt;<code>angle(8)=3*pi/2</code>&lt;br&gt;<code>angle(9)=2*pi</code></td>
<td>Define and initialize the array for all the angles of x (10 values)</td>
</tr>
<tr>
<td><code>for i = 0 to 9</code>&lt;br&gt;<code>...</code>&lt;br&gt;<code>next i</code></td>
<td>The i loop calculates all n values for each of the 10 angles.</td>
</tr>
<tr>
<td><code>for n = 2 to 10 step 2</code>&lt;br&gt;<code>...</code>&lt;br&gt;<code>next n</code></td>
<td>The n loop calculates all parts of the cos equation for each i angle</td>
</tr>
<tr>
<td><code>if n = 2 then</code>&lt;br&gt;<code>factorial = 2</code>&lt;br&gt;<code>temp(n) = 1 - angle(i)*n / factorial</code></td>
<td>Each n value (2, 4, 6, 8, 10) has different factorial values so calculate them individually.</td>
</tr>
<tr>
<td><code>if n = 4 then</code>&lt;br&gt;<code>factorial = 4 * 3 * factorial</code>&lt;br&gt;<code>temp(n) = angle(i)*n / factorial</code>&lt;br&gt;<code>...</code>&lt;br&gt;<code>if n = 8 then</code>&lt;br&gt;<code>factorial = 8 * 7 * factorial</code>&lt;br&gt;<code>temp(n) = angle(i)*n / factorial</code></td>
<td>If n is 4 or 8 then have positive fractions.</td>
</tr>
<tr>
<td><code>if n = 6 then</code>&lt;br&gt;<code>factorial = 6 * 5 * factorial</code>&lt;br&gt;<code>temp(n) = -1 * angle(i)*n / factorial</code>&lt;br&gt;<code>...</code>&lt;br&gt;<code>if n = 10 then</code>&lt;br&gt;<code>factorial = 10 * 9 * factorial</code>&lt;br&gt;<code>temp(n) = -1 * angle(i)*n / factorial</code></td>
<td>If n is 6 or 10 then have negative fractions.</td>
</tr>
<tr>
<td><code>cosofx(i) = cosofx(i) + temp(n)</code></td>
<td>Calculate the temporary cosofx per value of angle through using temp(n)</td>
</tr>
</tbody>
</table>

**PROGRAM 6**
<table>
<thead>
<tr>
<th>Source Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#define PI 3.14158</td>
<td>first I’ll define PI = 3.14158 I think - that’s close enough</td>
</tr>
<tr>
<td>» float power(float input, int x)</td>
<td>I’ll write a function for the power of</td>
</tr>
<tr>
<td>{</td>
<td>guess we need a float</td>
</tr>
<tr>
<td>» float result=input;</td>
<td>we’ll take an integer argument - call it x</td>
</tr>
<tr>
<td>» int c;</td>
<td>define a counter, c</td>
</tr>
<tr>
<td>» for (c=0; c&lt;x-1; c++)</td>
<td>for c=0; c&lt;x; c++</td>
</tr>
<tr>
<td>result *= input;</td>
<td>I better define a float for the result</td>
</tr>
<tr>
<td>return result;</td>
<td>okay so in the loop result = result * x</td>
</tr>
<tr>
<td>}</td>
<td>I think that’s all we need, and then return result</td>
</tr>
<tr>
<td>» float factorial(int x)</td>
<td>okay now I’ll do a function for factorial that will also be a float, I guess</td>
</tr>
<tr>
<td>int c;</td>
<td>- call it factorial</td>
</tr>
<tr>
<td>» float result=0.0;</td>
<td>take an integer argument, define a counter, have a loop</td>
</tr>
<tr>
<td>» for (c=1; c&lt;x; c++)</td>
<td>float result</td>
</tr>
<tr>
<td>» result *= x;</td>
<td>result=result+x</td>
</tr>
<tr>
<td>» return result;</td>
<td>c=0; c&lt;x; result=result*x; so we better start c at 1;</td>
</tr>
<tr>
<td>}</td>
<td>start c at 1 because if we multiply by zero the result will always be zero</td>
</tr>
<tr>
<td>void main()</td>
<td>we start c at 1 because if we multiply by zero the result will always be</td>
</tr>
<tr>
<td>{</td>
<td>zero we start c at 1 because if we multiply by zero the result will always</td>
</tr>
<tr>
<td>» float input[] = { 0, PI/6, PI/5, PI/4, PI/3,</td>
<td>zero we start c at 1 because if we multiply by zero the result will always</td>
</tr>
<tr>
<td>» float output, frac;</td>
<td>zero we start c at 1 because if we multiply by zero the result will always</td>
</tr>
<tr>
<td>» int power=2, count=0;</td>
<td>zero we start c at 1 because if we multiply by zero the result will always</td>
</tr>
<tr>
<td>» bool add;</td>
<td>zero we start c at 1 because if we multiply by zero the result will always</td>
</tr>
<tr>
<td>» while (count &lt; 10)</td>
<td>zero we start c at 1 because if we multiply by zero the result will always</td>
</tr>
<tr>
<td>» output=1.0;</td>
<td>zero we start c at 1 because if we multiply by zero the result will always</td>
</tr>
<tr>
<td>» add=FALSE;</td>
<td>zero we start c at 1 because if we multiply by zero the result will always</td>
</tr>
<tr>
<td>power=2;</td>
<td>zero we start c at 1 because if we multiply by zero the result will always</td>
</tr>
<tr>
<td>}</td>
<td>zero we start c at 1 because if we multiply by zero the result will always</td>
</tr>
<tr>
<td>» while (power &lt;= 10)</td>
<td>zero we start c at 1 because if we multiply by zero the result will always</td>
</tr>
<tr>
<td>» frac =</td>
<td>zero we start c at 1 because if we multiply by zero the result will always</td>
</tr>
<tr>
<td>powerof(input[count], power) /</td>
<td>zero we start c at 1 because if we multiply by zero the result will always</td>
</tr>
<tr>
<td>factorial(power);</td>
<td>zero we start c at 1 because if we multiply by zero the result will always</td>
</tr>
<tr>
<td>if (add)</td>
<td>zero we start c at 1 because if we multiply by zero the result will always</td>
</tr>
<tr>
<td>output = output + frac;</td>
<td>zero we start c at 1 because if we multiply by zero the result will always</td>
</tr>
<tr>
<td>else</td>
<td>zero we start c at 1 because if we multiply by zero the result will always</td>
</tr>
<tr>
<td>output = output - frac;</td>
<td>zero we start c at 1 because if we multiply by zero the result will always</td>
</tr>
<tr>
<td>» power += 2;</td>
<td>zero we start c at 1 because if we multiply by zero the result will always</td>
</tr>
<tr>
<td>}</td>
<td>zero we start c at 1 because if we multiply by zero the result will always</td>
</tr>
<tr>
<td>» printf(&quot;This is the output: %f&quot;, output);</td>
<td>zero we start c at 1 because if we multiply by zero the result will always</td>
</tr>
<tr>
<td>» count++;</td>
<td>zero we start c at 1 because if we multiply by zero the result will always</td>
</tr>
<tr>
<td>» }</td>
<td>zero we start c at 1 because if we multiply by zero the result will always</td>
</tr>
</tbody>
</table>
fraction, so I'll define a float called frac and we'll say frac = what's our x? OK, input count to the power of power; result = input - I'm looking up in the powerof function again - so I figure out the power of... okay and I divide that by the factorial of the same number - factorial of power.
so at this point frac = the fraction for the current power, and power starts at 2 and it needs to be incremented by 2; power +=2 and we go through up to and including 10. After this loop, output contains the result. I think I'll print it... printf this is the output %f, output And then end the outer loop, which will go through the input array Increment count at the end of the loop, while count < 10, output gets reinitialized, add gets reinitialized... that should be it.

### float powerof(float input, int x)

I'll write a function for the power.

```c
float result = input;
int c;
for (c=0; c<x-1; c++)
    result *= input;
return result;
```

Okay so in the loop result = result * x I think that's all we need, and then return result

### float factorial(int x)

Okay now I'll do a function for factorial that will also be a float, I guess - call it factorial

```c
float result = 0.0;
for (c=1; c<=x; c++)
    result *= x;
```

We start c at 1 we go up to and including x and each time through the loop we multiply result by the current x, so the end result should be factorial.

### float input[] = { 0, PI/6, PI/5, PI/4, PI/3, PI/2, 3*PI/4, 3*PI/2, 2*PI };

We'll define an array of floats, call it input and that will be our array of input values - and that's a constant array.
```plaintext
int power=2, count=0;

I'll define an integer called power that will represent the current exponent for x... initialize that to 2 and I'll also have an integer counter initialized to zero that will be an index into the input array.

while (count < 10)
We have 10 input values in the array, so while count is less than 10 loop

output=1.0;
add=FALSE;
output gets reinitialized, add gets reinitialized...

output=1.0;
We always start with 1 so I'll make it equal to 1 initially

add= FALSE
I need to figure out if I'm adding or subtracting, so let's define a boolean variable for that - boolean addsub, and we'll say if it's true we add - I'll call it boolean add - if it's true we add, if it's false we subtract; the first one we want to subtract so I'll initialize it to false.

if (add)
output = output + frac;
else
output = output - frac;
So in the loop, if add output = output + fraction, else output = output - fraction.

power += 2;
so at this point frac = the fraction for the current power, and power starts at 2 and it needs to be incremented by 2; power +=2 and we go through
```

### Table

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int power=2, count=0;</td>
<td>I'll define an integer called power that will represent the current exponent for x... initialize that to 2 and I'll also have an integer counter initialized to zero that will be an index into the input array.</td>
</tr>
<tr>
<td>while (count &lt; 10)</td>
<td>We have 10 input values in the array, so while count is less than 10 loop</td>
</tr>
<tr>
<td>output=1.0; add=FALSE;</td>
<td>output gets reinitialized, add gets reinitialized...</td>
</tr>
<tr>
<td>output=1.0;</td>
<td>We always start with 1 so I'll make it equal to 1 initially</td>
</tr>
<tr>
<td>add= FALSE</td>
<td>I need to figure out if I'm adding or subtracting, so let's define a boolean variable for that - boolean addsub, and we'll say if it's true we add - I'll call it boolean add - if it's true we add, if it's false we subtract; the first one we want to subtract so I'll initialize it to false.</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>if (add) output = output + frac;</td>
<td>So in the loop, if add output = output + fraction, else output = output - fraction.</td>
</tr>
<tr>
<td>else output = output - frac;</td>
<td></td>
</tr>
<tr>
<td>power += 2;</td>
<td>so at this point frac = the fraction for the current power, and power starts at 2 and it needs to be incremented by 2; power +=2 and we go through</td>
</tr>
</tbody>
</table>
Verbal Source Code Descriptor

<table>
<thead>
<tr>
<th><code>while (power &lt;=10)</code></th>
<th><code>After this loop, output contains the result.</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{</code></td>
<td></td>
</tr>
<tr>
<td><code>...</code></td>
<td></td>
</tr>
<tr>
<td><code>}</code></td>
<td></td>
</tr>
<tr>
<td><code>printf(&quot;This is the output: %f\n&quot;, output);</code></td>
<td></td>
</tr>
<tr>
<td><code>count++;</code></td>
<td><code>Increment count at the end of the loop</code></td>
</tr>
<tr>
<td><code>while (count &lt; 10)</code></td>
<td><code>And then end the outer loop, which will go through the input array</code></td>
</tr>
<tr>
<td><code>{</code></td>
<td></td>
</tr>
<tr>
<td><code>...</code></td>
<td></td>
</tr>
<tr>
<td><code>}</code></td>
<td></td>
</tr>
</tbody>
</table>
References


Verbal Source Code Descriptor


