University of Montana

ScholarWorks at University of Montana

Graduate Student Theses, Dissertations, & Professional Papers

Graduate School

1991

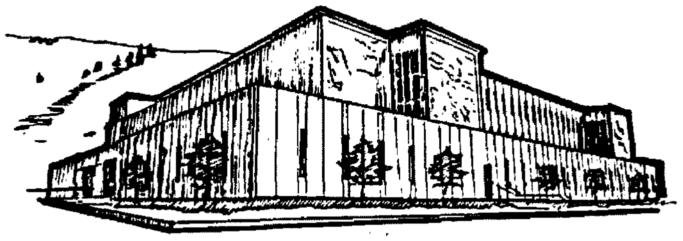
Textual display program for the dyslexic: An example of object-oriented software development

Joyce H. Fu
The University of Montana

Follow this and additional works at: https://scholarworks.umt.edu/etd
Let us know how access to this document benefits you.

Recommended Citation
Fu, Joyce H., "Textual display program for the dyslexic: An example of object-oriented software development" (1991). *Graduate Student Theses, Dissertations, & Professional Papers*. 7765.
https://scholarworks.umt.edu/etd/7765

This Thesis is brought to you for free and open access by the Graduate School at ScholarWorks at University of Montana. It has been accepted for inclusion in Graduate Student Theses, Dissertations, & Professional Papers by an authorized administrator of ScholarWorks at University of Montana. For more information, please contact scholarworks@mso.umt.edu.



Maureen and Mike
MANSFIELD LIBRARY

Copying allowed as provided under provisions of the Fair Use Section of the U.S. COPYRIGHT LAW, 1976.
Any copying for commercial purposes or financial gain may be undertaken only with the author's written consent.

University of
Montana

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A TEXTUAL DISPLAY PROGRAM FOR THE DYSLEXIC, AN EXAMPLE OF OBJECT-ORIENTED SOFTWARE DEVELOPMENT

Joyce H. Fu

B.S., Beijing Computer Institute, 1983
M.S., University of Montana, 1991

Presented in partial fulfillment of the requirements

for the degree of

Master of Science

University of Montana

1991

Approved by

[illegible]

Chairman, Board of Examiners

[illegible]

Dean, Graduate School

March 28 1991

Date

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

UMI Number: EP38566

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI
Dissertation Publishing

UMI EP38566

Published by ProQuest LLC (2013). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code

ProQuest

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Fu, Joyce H., M.S., March. 1991 Computer Science

A Textual Display Program For The Dyslexic,
An Example of Object-Oriented Software Development (69+v pp.)

Director: Alden H. Wright [illegible]

According to the latest statistics, about six to eight percent of the people in the U.S. have learning disabilities. Dyslexia is one of the major causes of these disabilities. Dyslexic people have visual perception problems which cause many difficulties in life, especially with reading.

Current computer applications extend more choices to the disabled learner than the traditional methods. Computerized books and speech synthesizer together provide a helpful learning environment for the visually disabled reader.

This project's goal is to create a textual display program whereby the user alters the display characteristics to potentially enhance dyslexic readers' reading ability. Using this program, the dyslexic may assign different colors to different letter combinations, change the distance between lines and/or words, change letter font or size. The program is developed with Actor, an object-oriented programming language, and therefore, this project is also an experience in object-oriented software development for the author.

This program focuses on the idea of helping the dyslexic readers toward a more practical reality rather than simply producing a software product. Although the program has not yet been used by dyslexics, it does achieve the requirements specified in chapter two of this paper. Therefore, the author believes the program to be not only a good start toward textual clarity but also a useful tool in the future for dyslexic readers. Finally, the author anticipates this software will be a useful tool for the educational researcher who desires to formally test the hypothesis that this type of program can help dyslexics.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Acknowledgments

With special thanks to Dr. Alden H. Wright of the Computer Science Department, George Kerscher of the Computerized Books for the Blind and everyone who gave me support through the entire process.

This effort is dedicated to my parents and to the memory of my grandmother.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table of Contents

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter One

Introduction

1.1. Project Background

In general, a person of average or above average intelligence, who performs at a level two years behind his/her normal grade-level expectancy, is considered to be learning disabled. According to the latest statistics [Bureau of Statistics, Washington D.C. 1989], about six to eight percent of the people in the U.S. have a learning disability. Dyslexia is one of the major causes of learning disabilities. Dyslexic people have visual perception problems which cause many difficulties in life, especially with their reading.

The traditional methods of providing access to standard print for those with visual perception problems include large print versions, recordings, and braille. However, current computer applications can extend more choices to the disabled learner. For example, computerized books (software versions of books) provide dyslexic readers with the ability to read books from the computer screen. Adaptive computer technology such as speech synthesizers for the blind, telecommunications devices for the deaf, voice recognition, and other control devices for the motor-impaired, offer greater opportunities to people afflicted with either sensory or physical disabilities. The speech synthesizer is one of the most powerful and least expensive access devices. It converts ordinary ASCII text into intelligible speech. Together, computerized books and speech synthesizers provide a helpful learning environment for the visually disabled reader. For

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

instance, today's screen-review software couples the reader and the speech synthesizer, allowing the dyslexic to depend upon auditory output to overcome his/her reading disability when confronted with confusing textual displays.

1.2. Project Idea

In 1989, Disabled Student Services Counselor, Christie Horn, discovered that students with dyslexia could read more effectively when using word processors with modified colors and in some cases enlarged text [Horn 1989]. Some students required only a change of background and foreground color to make the text comprehensible, and others needed the additional enlarged characters to assist them in their word processing endeavors. These observations now lead reading specialists to speculate that, in some cases, dyslexics may gradually overcome their difficulties through training and eventually achieve normal reading ability [Horn 1989, Jakupcak 3/1990-3/1991 and Kerscher 3/1990-3/1991]. In order to overcome reading deficiency and train toward reading proficiency, dyslexics need not only to hear from a speech synthesizer but also to actively manipulate the textual display. Therefore, this project's major concerns are to enable the dyslexic to easily manipulate the textual display on a graphics screen for enhanced readability and to provide the dyslexic with a supplementary speech synthesizer capability.

1.3. Project Overview

1.3.1. Goal of the Software Project

Based on Horn's discoveries, reading specialists now hypothesize that further changes to the display of text are a possible way to help the dyslexic read more

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

easily from a computer screen. Using this hypothesis, the primary goal of this software project is to create a textual display program whereby the user alters the display characteristics to potentially enhance his/her ability to read the text. Horn's findings inspire the use of more colors, fonts, and other alterable features in the display of text. Thus, this software allows the dyslexic user to decide the way in which the text will appear on the screen. Using this program, the dyslexic may assign different colors to different letter combinations, change the distance between lines and/or words, change letter font or size, etcetera. He/she also has the option of using a speech synthesizer. Overall, this software should enable the dyslexic to modify the display of text on the screen as desired while maintaining the supportive aid of a speech synthesizer. The author anticipates that this software will be a useful tool for the educational researcher who desires to formally test the aforementioned hypothesis.

1.3.2. Process of Software Development

The choice of a software development methodology involves four interdependent choices: Specification methodology, design methodology, type of display, user-interface, and programming language.

Specification methodology

To choose a specification methodology, the developer has at least three choices: Narrative English, Structured Analysis [Demarco 1979, 1978] and Object-Oriented Analysis (OOA) [Coad and Yourdon 1990]. Narrative English uses mainly English sentences to describe the user's problems and needs and to describe the system specifications. This is the simplest methodology and maybe a good choice for a small system. Both structured analysis and OOA use diagrams, dictionaries, and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

other tools to aid the developer in defining and analyzing the problems for a large complex system. Since this project does not deal with a complicated system, structured analysis or OOA is not necessary. Thus, the author chose narrative English as the appropriate method for specification of this particular software development project.

Design methodology

Two of the important software design techniques are: Structured Design [Yourdon and Constantine 1979] (also called traditional design) and Object-Oriented Design (OOD) [Booch 1991]. Structured design, a process of top-down functional decomposition, is commonly used in software development. The OOD technique applies object decomposition through the design process. It is not only a new technique to most developers, but also maybe a better technology in terms of software quality (especially for a complex system). In order to try out this new technique, this project used OOD as the design methodology. More discussion about OOD will follow in a later chapter.

Type of display

There are two kinds of displays: the standard character display which allows only textual display, and the graphical display which handles both text and graphics. Considering this program's changes to line space, letter size and font etc. (further explained in chapter two), the standard character display is not sufficient. However, the graphical display is capable of handling font and letter size etc. and so meets this project's display requirements.

User-interface

One of the most complicated programming areas is user-interface development.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

There are many kinds of user-interfaces based on different types of display: Vi 3.5/2.13 [University of California 1979 1980] which provides a character user-interface, WordPerfect 5.0 [WordPerfect Corp. 1982 1989] which utilizes function-key interface and Turbo C++ 1.01 [Borland International 1987 1988] which contains a windows interface with menus, dialogues, scroll-bars etc.. However, these three are all based on a standard character display and do not support graphical displays. On the other hand, MS-Windows 3.0 [Microsoft Corp. 1985-1990], a graphical user-interface, supports both standard character display and graphic displays, providing an easy-to-use and a consistent graphics windowing environment. Therefore, MS-Windows' user-interface was the final choice.

Programming language

The choice of a specific programming language is obviously another important factor in the development of a software project. Quite a few alternatives are available for Microsoft Windows development. They include: C or C++ supplemented with the Microsoft Windows Software Development Kit (SDK) [Microsoft Corp. 1990], ToolBook 1.0 [Asymetrix Corp. May 1990], KnowledgePro 1.0 [Knowledge Garden Inc. June 1990], Smalltalk-80 [Xerox Corporation 1983] and Actor 3.0 [Whitewater Group 1990].

The Windows Software Development Kit (SDK) is a tool for the programmer who is developing a MS-Windows application. It enables the user to interact directly with a windows programming environment. SDK provides a set of C run-time libraries which are compatible with a Microsoft C compiler and Microsoft Windows. The development kit enables the programmer to use dialogues, menus, and other windows resources, and provides debugging tools, sample source code, on-line references, and so on. However, since SDK is not an object-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

oriented library, the programmer still faces a heavy C or C++ coding task with SDK.

In addition, two object-oriented development tools: ToolBook and KnowledgePro for applications in MS-Windows 3.0 might be feasible tools for this program, but they were not available at the time this project started.

Smalltalk-80 is an object-oriented programming language and also a windowing environment. It now has software to support MS-Windows programming. However, at the time this project started, the supportive MS-Windows software has not been released. Thus, under the choice of MS-Windows as the user interface, Smalltalk-80 was not a feasible language for this project.

Actor is an object-oriented language created for MS-Windows programming. It provides a rich class library to manage graphical user-interface programming and enables most of MS-Window's features. Actor benefits the programmer with both simplified windows programming and maximum usage of windows features. Another influential consideration is that Actor is priced inexpensively for academic institutions. For these reasons, Actor was selected as the programming language for this project.

The discussion above describes four methodology choices for this project. They are different concepts but are interdependent. This is especially evident among the design, user-interface and programming language. The OOD choice specifies that an OOP language. With the choice of the MS-Windows user-interface, Actor became the appropriate OOP language.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.4. Object-Oriented Programming (OOP)

Although it has been almost two decades since the idea of Object-Oriented Programming came up for the first time [Bobrow 1989], it is still a relatively new concept to most programmers. In the following sections, a brief introduction of the OOP concept and a description of the OOP terminology will be discussed.

1.4.1. The Concept

"*Object-oriented* means that we organize software as a collection of discrete objects that incorporate both data structure and behavior" [Rumbaugh et al. 1991]. The term **object** refers to the combination of data and procedures in the OOP concept. Object-Oriented Programming is a conceptual model for software development. **Modularity** (well packaged data and procedures that operate on the data) is the key to improving software's reusability and extendibility. Encapsulation and inheritance are the two OOP features which enhance software with respect to modularity, resulting in higher quality software.

Encapsulation (also known as information hiding) is generally used to describe an object's protection of its private data from outside access. In other words, data is packaged only with the procedures that access that data. This concept is often called data abstraction or modular programming [Thomas 1989].

Inheritance centers on the object construction and is a fundamental technique for both reusability and extendibility. Each object is derived from a corresponding class. Objects inherit both data and behaviors from their parents and may also own their private data. In a class hierarchy, a lower-level class inherits both

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

instance variables and methods that are have been defined in its parent classes. Since inheritance allows partial modification to software, it not only simplifies the program but also provides possibilities of reuse and extension.

The above description introduces the OOP concept through its features. It illustrates that OOP encourages programming in terms of "objects", rather than separate "procedures" and "data." Compare with structured technique, the difference is obvious: data are no longer separated from the procedure under the OOP paradigm.

1.4.2. Terminology

The idea of OOP consists of four major components: <u>Object</u> (the crucial component), <u>Class</u>, <u>Method</u>, and <u>Instance variable</u>. The following definitions for these terms will further explain the concept of OOP.

<u>Object</u>
"The initial appearance of the notion of an 'object' as a programming construction was in Simula, a language for programming computer simulations." [Birtwistle et al. 1973] One may ask two questions about "object": What is an object? What is in the object? The answer to the first question is that "an **object** is an entity in the real world." In other words, we perceive the world around us as a variety of objects. For example, a person is an object, a picture is an object, and a "**specific**"**-car** is also an object. Each object contains its own characteristics along with a set of manipulations on itself; this is the answer to the second question. Using the car example given above, a car has its manufacture-name, model-type, color and a series of functional behaviors. A more precise definition

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

with technical terms of "object" is as follows: "A data structure along with all the procedures that operate on it is bundled together in a distinct module called an object. Similarly, an 'object' is a self-contained entity which has its own private data and a set of operations to manipulate that data. One can also say, an "object" is a block of code that contains a data structure together with all of the procedures that are required to operate on the data elements of the data structure. Therefore, an object is capable of receiving, understanding, and implementing high level "messages" from other objects.

Class

The class concept is central to OOP. Generally speaking, a class is a set of similar objects. More specifically, a **class** is a template for creating individual objects that behave in a similar manner. Each object belongs to a class that defines the type of the object. **Car** is a class for the car example given above. Objects of this class may include *blue-91-Toyota-Camry* and *red-88-Ford-Taurus* . One can also say that an object is an instance of its class. Different classes are related according to an inheritance hierarchy. One ancestor class might give rise to many descendant classes, which might, in turn, be ancestors of yet other classes. The procedures and data structures are passed along to descendants. This is called inheritance. For example, the **Car** class and the **MotorBoat** class can both be descendant of the **MotorizedVehicle** class, and both inherit data and procedures from the **MotorizedVehicle** class. Two kinds of inheritance exist: single inheritance and multiple inheritance. Under single inheritance, a class is only allowed to have a single parent class. Multiple inheritance allows a class to have more than one direct parent. Inherited behavior is used as the default if the behavior is not redefined in the descendant. As mentioned before, inheritance is one of the significant OOP characteristics which distinguishes OOP from other techniques.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Method

A message to an object is a request to execute a specific function or algorithm. Such a message is called a **method**. Objects from the MotorizedVehicle class have behaviors: such as turn left, go-straight and stop. In other words, they can respond to "turn-left", "go-straight" and "stop" messages. The corresponding methods are defined in either the object's class definition or in the definition of its ancestor classes. One attribute of a method is *overloading* which means different classes may define methods of the same name. Therefore, objects of many classes can respond in their own appropriate ways to the same request. For example, the *turnLeft()* method may be defined differently in both Car and MotorBoat classes. Thus, a "specific"-car (instance of Car class) and a "specific"-motor-boat (instance of MotorBoat class) can respond to the same "turn-left" message in different way.

Instance variable

An instance variable is a variable contained in an object. Again using the above example, instance variables in the **Car** class could be: manufacture-name, model-type, color and so on. Instance variables may also be objects and can belong to any other class. This permits nested data structures.

The capabilities of objects, classes, methods and instance variables together make OOP a complete and powerful technique in the field of computer programming. Subsequent chapters cover the OOP design and implementation techniques used in the development of this software project.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter Two

Requirements and Specifications

To accomplish the goal of a software development project, the developer needs initially to clarify the software requirements and system specifications. Requirements and specifications are results of analyzing the user's needs. This chapter covers two major requirements and two major specifications: The first requirement includes textual information display on a computer screen with a variety of formats of space, color and font. The second requirement concerns use of the speech synthesizer. Specifications include the computer hardware, its operating system, speech synthesizer and input file format.

2.1. Text Display Requirements

Dyslexic individuals have various reading difficulties. Some of them have one or two specific problems, and others have more. Following are five specific problems which have been identified as potential problems for dyslexics [Horn 1989, Jakupcak 3/1990-3/1991 and Kerscher 3/1990-3/1991]. Although dyslexics have many additional problems that this paper does not cover, the following problems are sufficiently common to justify this program. In order to achieve the following requirements, the software requires the fundamental precondition of opening a text file for input.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Problem 1: The dyslexic may have problems separating lines. In other words, lines may be perceived as crossing in text with commonly used formatting.
Requirement 1: The software should allow for easy modification of spacing (measured in pixels) between lines.

Problem 2: For some dyslexics, words may perceptually overlap in a text with the usually formatting.
Requirement 2: The software should allow for the modification of spacing (measured in character width) between words.

Problem 3: The dyslexic may have problems distinguishing between a particular pair of letters (such as b's and d's). Similarly, the dyslexic may have trouble reading some letter combinations such as "ch," "the," or "l." Whenever these combinations appear in a text, the dyslexic reader may have a hard time perceiving these combinations.
Requirement 3: Since most dyslexic readers may distinguish colors correctly, the software should allow for color modification to letters so that all occurrences of the particular letter pairs or letter combinations are displayed with an assigned color. For example: The dyslexic reader may assign red color to "b," blue to "d," pink to "ch," and green to "the." All the occurrences of "b," "d," "ch" and "the" will then appear corresponding to a specific color. In addition, the dyslexic may choose to contrast background color against character color.

Problem 4: Some dyslexics may have difficulty in perceiving the first letter of each word.
Requirement 4: The software should enable readers to modify the color of each word, so that the first letter of each word will be displayed with a specific color

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

throughout the entire text.

Problem 5: The dyslexic reader may have problems reading the usual letter size of text.

Requirement 5: The software should enable the reader to modify the letter size, so the entire text displays with that specific letter size. In other words, the dimensions of letters can be adjusted to a bigger or smaller size. The result of this feature is similar to enlarged or reduced print on paper.

These five problems and the corresponding requirements describe common difficulties of dyslexic individuals, and the solutions that this software should provide in order to help. The following are the extended requirements which can be accomplished by the software. Some may prove useful for certain types of dyslexia.

Requirement 6: The software allows dyslexics to change the character's font. For instance, the reader can specify that the entire text be displayed in Roman, Modern, Courier and etcetera.

Requirement 7: The character style, including "italic," "bold face," and "underline," is also changeable. The reader may assign a specific style for the entire text, for the first letter of each word, or for certain letter combinations.

Requirement 8: The software also allow the dyslexic reader to reverse the text color and background color for the entire text. It will allow the reader to employ this feature on a specific letter pair, or the first letter of each word.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Requirement 9: Words in the text file can be traced with the use of the four arrow-keys and subsequently highlighted with reverse-video display. For example, when the user presses the "->" key on the key-board, the word to the right of the current word will be changed to a reverse-video display. Also, the software defines a set of keys to handle horizontal and vertical page movements.

2.2. Speech Output Requirements

A speech synthesizer used on a computer converts text into simulated speech. This software package will also allow the dyslexic to use a speech synthesizer. The following are four main concerns of using a speech synthesizer in terms of users' need.

First, the dyslexic reader may still have problems with reading even after modifying the textual display characteristics. For instance, the reader might have a particular problem which is not being taken care of through the display attributes provided by this software. He/she might want to get some help from some sort of speech device in order to overcome the problem. Therefore, the software should not only allow textual modification, but also allow the reader to employ a speech synthesizer while the text is being displayed.

Secondly, dyslexic individuals may prefer the synthesizer to spell out each letter, rather than pronounce the complete words, or vice versa. The software should give the reader two choices: spell out letter by letter, or read word by word. The reader should be able to convert this setting at will.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Third, the synthesizer can either sound out the whole text, or pronounce (or spell) words which the current cursor marks. The dyslexic may use four arrow keys (on the key board) to move the cursor around the text.

Finally, the dyslexic reader may not understand the speech with the default settings of speech attributes which include: voice, rate, pitch, intonation, etcetera. Thus, if the synthesizer permits, the software will allow the reader to reset those speech attributes at any time.

2.3. Hardware and Operating System Specifications

The program requires an IBM PC or compatible with a hard disk and a color monitor (EGA or VGA). The program also requires MS- Windows, version 3.0 or later. MS-Windows requires 640K bytes of memory. The performance of MS-Windows is better on a 286 or 386-based computer.

The speech synthesizer used in this program is called ACCENT Stand Alone (ACCENT-SA), which is a stand alone device for all computers. It has a 640-byte warp-around text buffer. A standard RS-232C serial port is provided for interface with a computer or terminal as the host. It responds to four groups of software commands: system option, speech option, index option, and status request.

The operating systems that support this software are those systems that support MS-Windows, currently MS-DOS and PC-DOS.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.4. Input File Requirements

The text file which can be displayed in this program is a DOS text file (also called an ASCII text file), with no special formatting characters, where each line is at most 80 characters long. A product of Computerized Books For the Blind and Print-Disabled should fit the above requirements.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter Three

Object-Oriented Design (OOD)

Design means to plan or mark out the form and method of solving a problem. Although there are many design techniques, problem decomposition is the key to all techniques. For example, the top-down structured design, used for a procedure-oriented style, approaches decomposition as a simple matter of algorithmic decomposition. The object-oriented technique, on the other hand, uses the technique of object decomposition rather than algorithmic decomposition. The following is the definition of OOD: "Object-oriented design is a method of encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design" [Booch 1991]. To apply the object-oriented technique in software development, the developer's major task is to build an *object model* to capture the static structure of the system. The definition of object model will be given in section 3.2.

3.1. Chapter Overview

This chapter gives an overview of the use of object-oriented design in this project. It explains, beginning with the initial class design and concluding with an outline of the entire display project's hierarchy and class relationships, the process of constructing an object model for this software. At the end of this paper, three appendices: Appendix A (class-behaviors-collaborators), Appendix B (method-description) and Appendix C (data-dictionary) further illustrate the detailed design

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

documentation for this project. The last section focuses on the design of the user-interface.

3.2. Object Model

"An *object model* captures the static structure of the system by showing the objects in the system, relationships between the objects, and attributes and operations that characterize each class of objects" [Rumbaugh et al. 1991}. Within the *object model* exist three important characteristics: abstraction, encapsulation and hierarchy.

Abstraction: Abstraction is one way to deal with complexity in the real world. An abstraction denotes the essential characteristics of an object that distinguish it from all others and thus provides crisply defined conceptual boundaries. Abstraction leads to an object model where each object has a greater cohesiveness but a looser coupling relationship to other objects.

Encapsulation: As mentioned in the first chapter, encapsulation means information hiding. It hides the internal implementation details of an object from other objects. By employing encapsulation, one can modify an object without affecting the applications that use the object. The two concepts, abstraction and encapsulation, are complementary to each other. To an object, abstraction focuses externally while encapsulation focuses internally.

Hierarchy: In the real world, objects are categorized or grouped into classes, and classes can be further grouped into super-classes. Those objects and classes form a hierarchy by their characteristics. Inheritance is the most important hierarchy. Using object-oriented design, proper use of inheritance not only

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

simplifies the understanding and implementation of the problem, but also enhances the software's reusability.

Both *object* and *class* are terms used in object-oriented programming, and sometimes what distinguishes an object from a class is confused. Therefore, it is very important to clarify these two terms before the object model construction process.

3.2.1. Object and Class

As mentioned in chapter one, object is an entity in the real world. To be more specific, there are two kinds of objects: physical entities which one can see in the real world, and concepts which are not visible. A class is a template for a set of similar objects and is an abstract data type implementation. Actually, a class is the technical term applied in object-oriented programming to describe the data structure and common behavior which characterize a set of similar objects. Objects and classes are separate yet intimately related concepts. Specifically, every object is the instance of some class, and every class has zero or more instances.

In an object-oriented programming environment, an **object** is dynamically created during a system's execution, whereas a **class** is a statically describes of a set of possible objects. Therefore, object-oriented design is actually embodied in class design. In other words, the conceptual framework of object-oriented design is to identify classes, state the relationships between classes, and define both data and operations which characterize each class of objects. In order to construct an

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

object model, the first step is to identify relevant classes from the problem domain.

3.2.2. Problem Domain

Problem domain is a term to describe the problem space in the real world. Looking at the entire problem from the top level: What is the problem? What is the boundary of the problem? What is the software's responsibility in terms of solving the problem? For example, the overall goal of the display software, as stated in the first chapter, is to both empower the user to change the display attributes of the text and to enable the use of a speech synthesizer if required. A context diagram (figure 3.1) is a good way to depict problem space. The diagram shows the program task, the interface with external entities and the problem boundary.

This particular problem contains four external entities: the user, the text file, the computer screen and the speech synthesizer. These entities' actions rely on the program as the central controller. Each entity has a relationship with the program. Initially, the user specifies a text file name for the program, and the program then opens the file and brings it up on the screen. Whenever the user changes display requirements, the screen receives commands with display attributes to show the newly specified text. Finally, if the user so desires, the speech synthesizer receives commands from the program and then either pronounces or spells out words. The context diagram clarifies what the program should and should not do. It needs to handle the input from the user, manipulate the display to the screen, and also operate the speech synthesizer. This is the domain of this particular problem.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

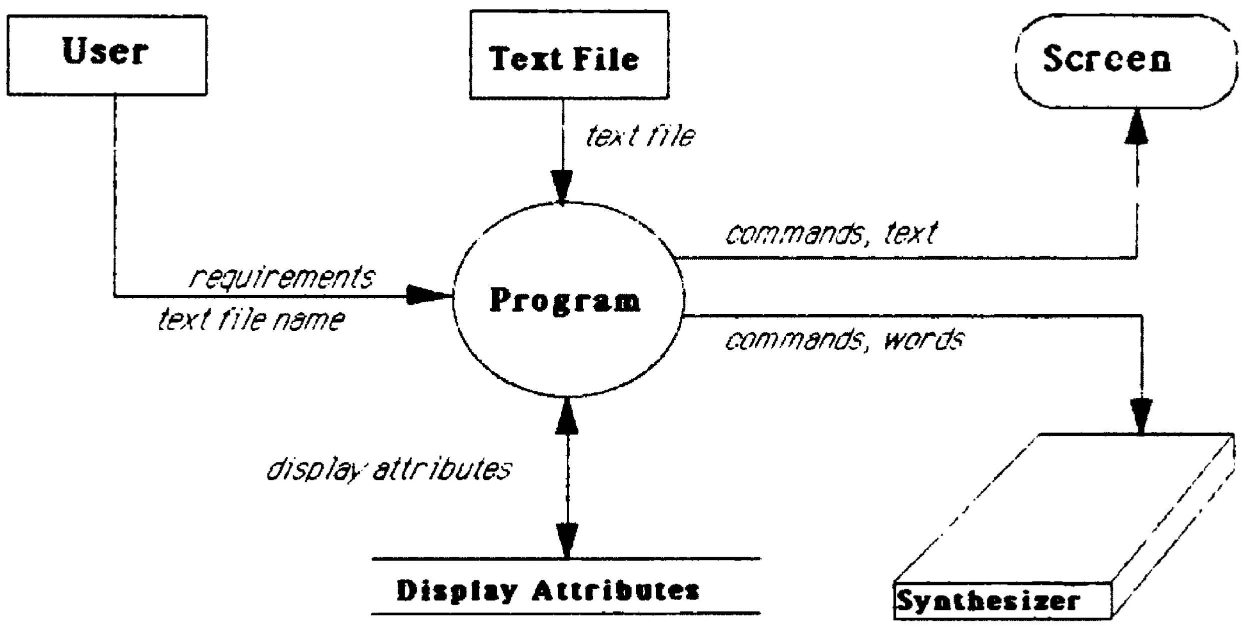


Figure 3.1: Context Diagram

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.2.3. Initial Class Design

Four classes can be initially identified from the above problem domain: text file (called DispCollection), display window (called DispWindow), synthesizer (called Synthesizer) and display attributes (called VisualAttributes). They each have responsibilities as described in table 3.1.

Class Name	Responsibilities
DispCollection	hold the text file as a collection of lines; seek words, letters etc. in the text.
DispWindow	draw text on the display window according to the visual attributes; control the speech synthesizer
Synthesizer	receive commands from the display window; connect serial port and send text to serial port
VisualAttributes	set visual attributes; provide different attributes to the display window for textual display

Table 3.1: Some major classes and their responsibilities

In addition, these four classes are interrelated. They collaborate with each other in the course of satisfying responsibilities. For instance, DispWindow collaborates with Synthesizer and other classes (Appendix A) to satisfy its responsibilities. Each class contains both data and behavior. Moreover, each class defines a set of instance variables and methods. For instance, the Synthesizer class defines the instance variables: sVoice, sRate, sPitch and etc. (Appendix C) to hold the speech attributes, and a set of methods: sVoice(), setVoice() and etc. (Appendix B) to set and return the value of these variables. To apply the object-oriented design technique, further decomposition is required.

In table 3.1, the VisualAttributes class carries the responsibility of setting and providing different display attributes for the display window, and of representing the visual property abstraction of the text display. However, since visual display

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

attributes include text attributes, letter combination attributes and first letter attributes, visual attributes can be further divided into text attributes, letter combination attributes and first letter attributes. Thus, three classes: TextAttributes, LetterCombAttributes and FirstLetterAttributes, are derived from the VisualAttributes class. Furthermore, these three classes are sub-classes which inherit from the VisualAttributes class. Therefore, these sub-classes may have their own private data and behaviors while sharing basic data and behaviors from the VisualAttributes class.

Table 3.2 describes the inheritance of the attributes classes. For example, although no instance variables are defined in the TextAttributes class, instance variables are inherited from the VisualAttributes class instead such as: vBkColor, vFontColor. Methods inheritance applies to the TextAttributes class in a similar manner. The TextAttributes class defines its private methods (such as setLineSpace(), setWordSpace(); ...), and in addition inherits setBkColor(), setFontColor() and etc. methods from the VisualAttributes class.

Class Name	Responsibilities	Instance Vars.	Methods

VisualAttributes	initialize visual attributes; set and provide visual attributes to the display window	vBkColor;
		vFontColor;
		vUnderline;
		...
			setBkColor();
			setFontColor();
			setUnderline();
			...

TextAttributes	set and provide attributes for the entire text;
	especially set and provide the line-space and word-space values
			setLineSpace();
			setWordSpace();
			getFonts();
			applyNewFonts();
			...

LetterComb-Attributes	set and provide attributes for different letter combinations;	reverseFirst;
		...
			getFonts();
			applyNewFonts();
			...

FirstLetterAttributes	set and provide attributes for the first letter of each word	letterCombDict
		...
			getFonts();
			applyNewFonts();
			...

Table 3.2: Class inheritance for the visual attributes classes

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Three tables: class-behaviors-collaborators, method-description and data-dictionary, describe in detail the class design, data structure and the method design respectively for the entire display program (see Appendices).

3.2.4. Class Hierarchy

The class hierarchy depends heavily upon the built-in class library of a certain programming language. To properly place a newly created class into the existing class hierarchy is very important. For instance, in terms of hierarchy, where should the class DispWindow be placed into the Actor class library? Although each different programming language provides a different class library with a different class hierarchy, a clear understanding of each class is always the key. Consider the behavior of this particular window. It is a window, of course, and it allows text to be displayed but not modified in it. In the Actor class library, the window branch contains WindowObject, Window, TextWindow, EditWindow etc. in top-down order. Obviously, the display window should be placed under the class Window since it acts as a window. As the display window does not allow any modifications to it, one may quickly decide to make it a descendant of TextWindow class. This may be correct theoretically but possibly inappropriate pragmatically. Merely understanding the particular problem is not sufficient. One has to know the characteristics of related, built-in classes to be able to create the proper hierarchy. Returning to the example above, the DispWindow class actually needs most EditWindow services, except the editing behavior. For example, the user may insert, delete or modify characters to the text in an edit window. To make DispWindow a descendant from the EditWindow class rather than from the TextWindow class avoids definition redundancy. In addition, just one short method is required to disable text modification, and this decision produces a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

more appropriate class hierarchy. Figure 3.2 shows all of the classes for this project and their parent classes that exist in the Actor class library.

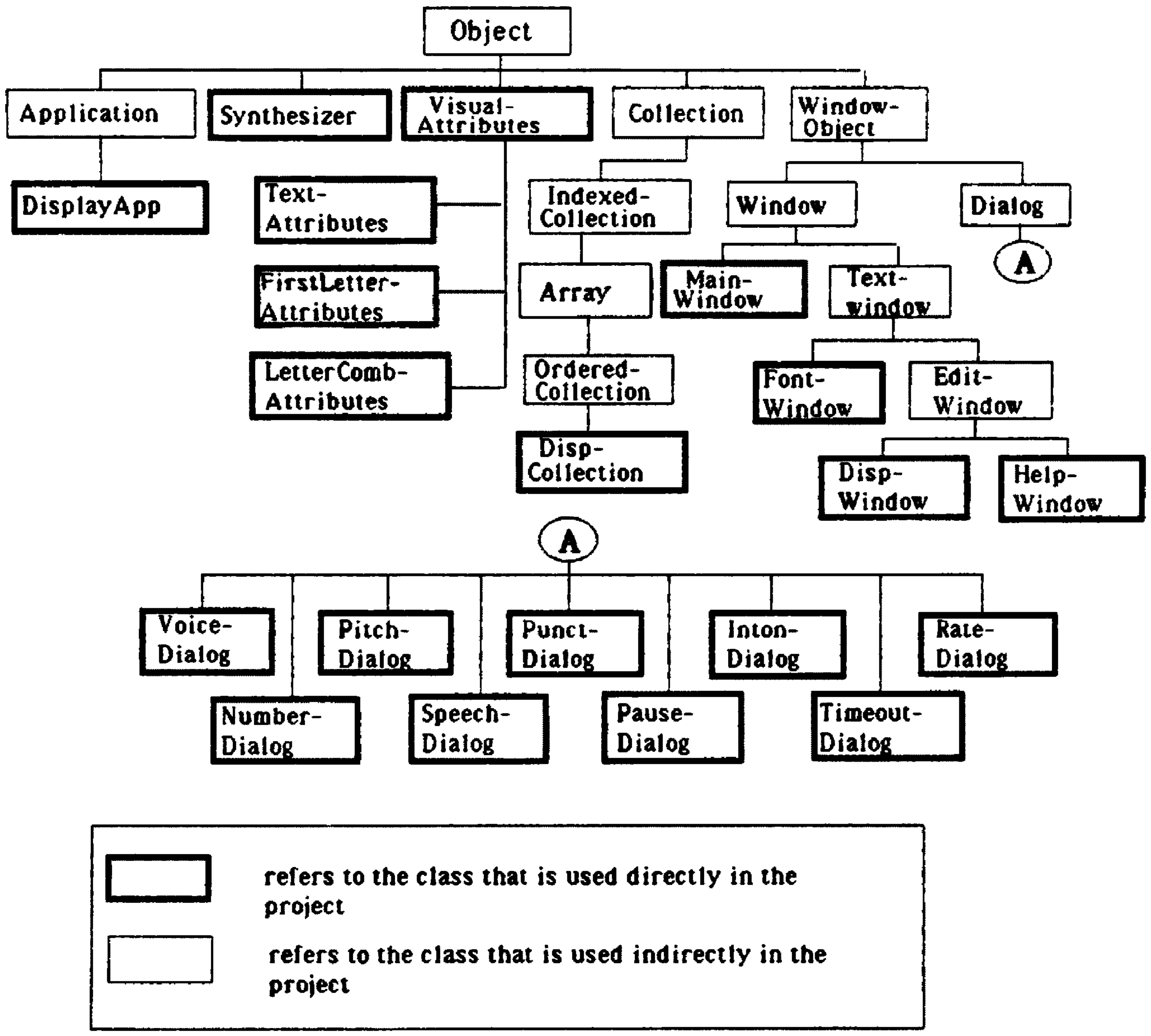


Figure 3.2: Class inheritance hierarchy

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.2.5. Relationship Between Classes

It is clear that a real world system is not a collection of a number of unrelated objects. Classes contribute to the behavior of a system by cooperating with one another. Two major kinds of relationships exist between any two classes: Using or inheritance.

<u>Using relationship</u> If a class plays an active or passive role, which means that it uses or can be used by other classes, then the relationship among them is a *using relationship* . For example, in the display program, DispWindow has a using relationship with Synthesizer. Whenever the user wants to hear from the speech synthesizer, the display window uses the synthesizer to handle the speech. Therefore, the DispWindow class operates upon the Synthesizer class. In this case, DispWindow is the actor, and Synthesizer has the supporting role.

<u>Inheritance relationship</u> If a class shares data structure and behaviors from another class, then these two classes are in an *inheritance relationship* . Table 3.2 reveals the inheritance relationship between some of the classes from the display software. TextAttributes, LetterCombAttributes and FirstLetterAttributes all share common data and behaviors with VisualAttributes apart from their private data and behaviors.

Figure 3.3 shows the complete view of all class relationships in the display software program.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

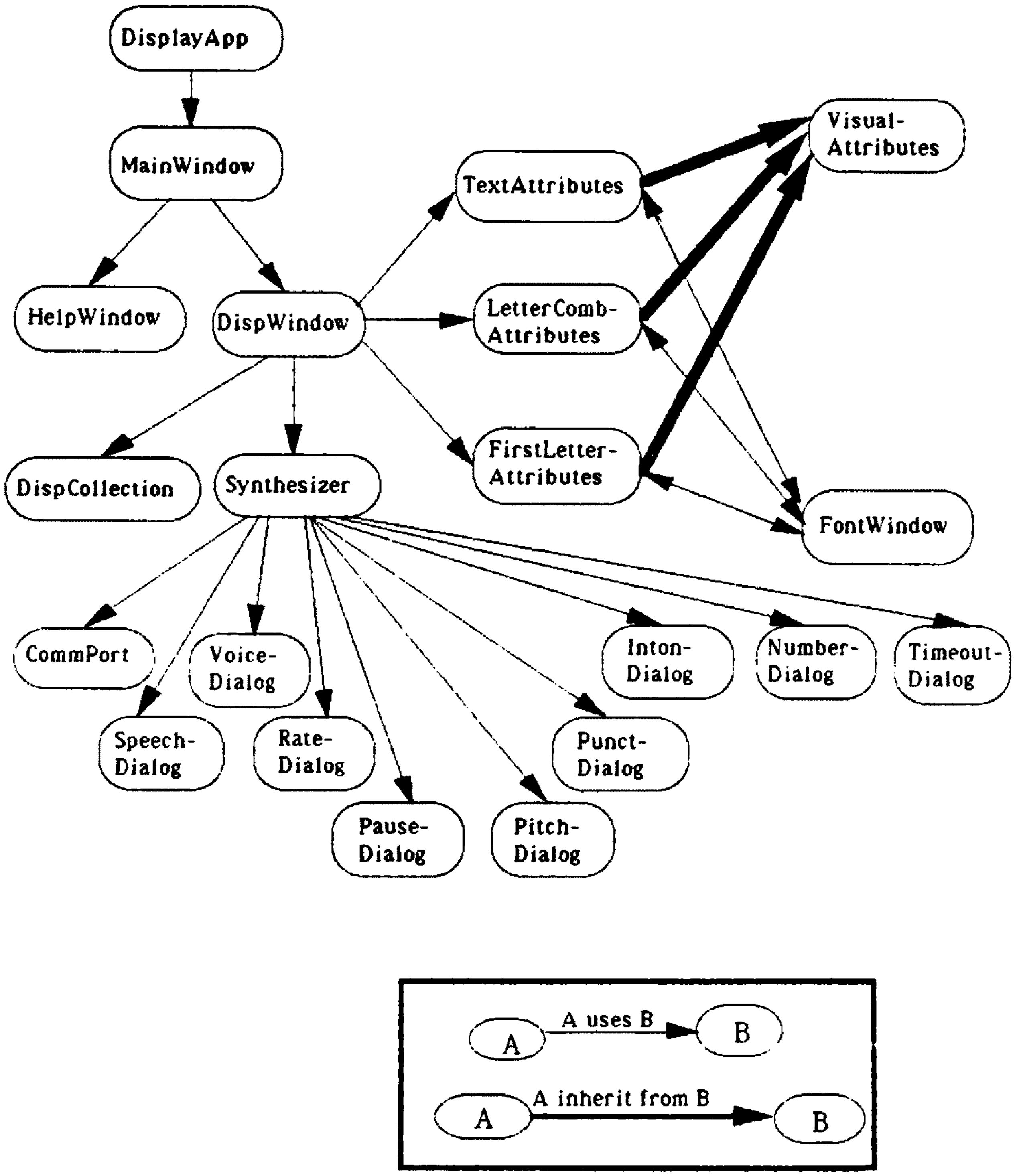


Figure 3.3: Class Relationships

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.3. User-Interface Design

The user-interface is the direct communication of a program with the user. It is a medium used for the program to receive the user's responses, choices or input data. The objective of user interface design is to provide users with the most reasonable presentation of the system's state and its functions. The difficulty of user-interface design is that it requires a fair bit of creativity to decide on a good paradigm for both presentation and manipulation. Fortunately, the task of the user-interface design for this software project is not too difficult. Actor, the programming language used in this project, enables the programmer to easily manipulate the MS-Windows' user-interface.

A window (figure 3.4) is the major interface in windows application. It has a title-bar, menus on a menu-bar and a view area. Windows provide two kinds of menu items on the menu-bar: <u>pop-up</u> menu item and <u>action</u> menu item. When selected, a pop-up menu item derives another menu with more than one option (figure 3.5). An action menu item directly performs an action once it is selected. Two kinds of dialogue-boxes are involved in this project: a selection dialog-box (figure 3.6) where the user can select from the given choices, and an input dialog-box (figure 3.7) where the user can type in data in the blank-space. A list-box (figure 3.8) is a dialogue-box with a vertical scroll bar which provides the user with a list of files to choose from. The user may select any file from the list-box, and read it into the display window. These three major user-interface components provide the user with a consistent and convenient means of communication to the program. Figure 3.9 gives the top-down user interface hierarchy of the display software. For instance, the main window provides three action menu items: CloseMainWindow, OpenDispWindow and OpenHelpWindow. Once the user selects the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

OpenDispWindow item, the display window shows on the screen with five pop-up menu on its menu-bar.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

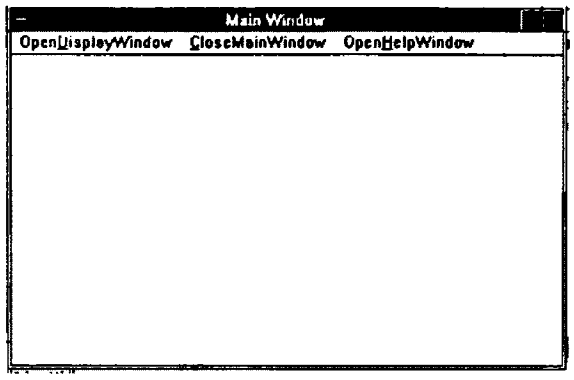


Figure 3.4 A Window

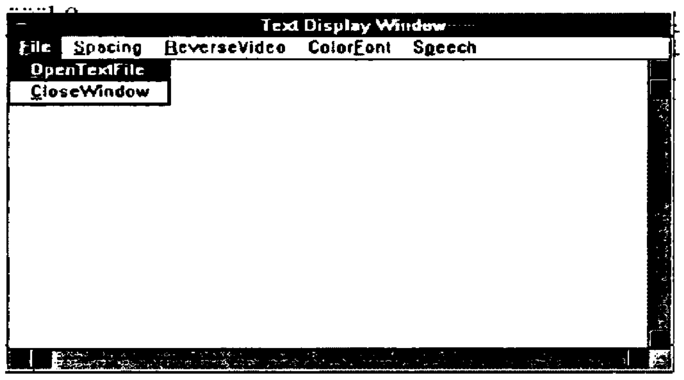


Figure 3.5 A window with a pop-up menu

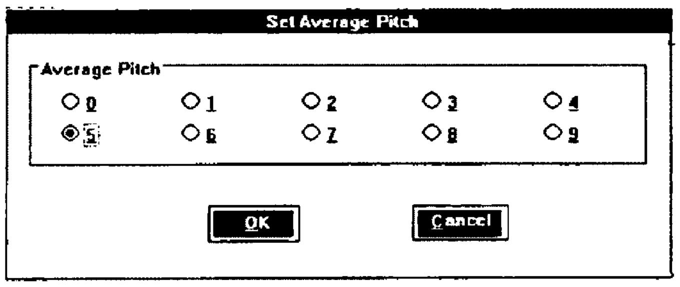


Figure 3.6 A selection dialog-box

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

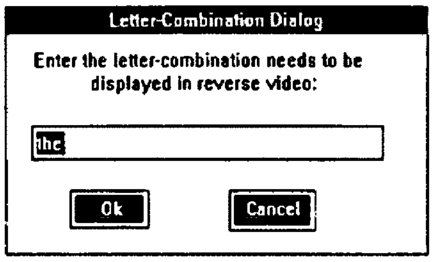

Letter-Combination Dialog
Enter the letter-combination needs to be displayed in reverse video:
the
Ok
Cancel

Figure 3.7 An input dialog-box

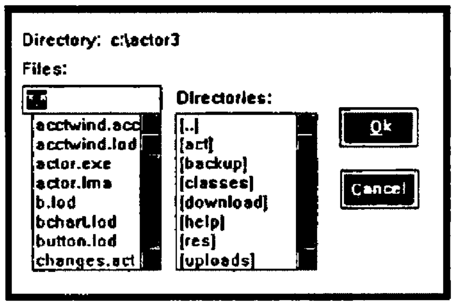


Figure 3.8 A list-box

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

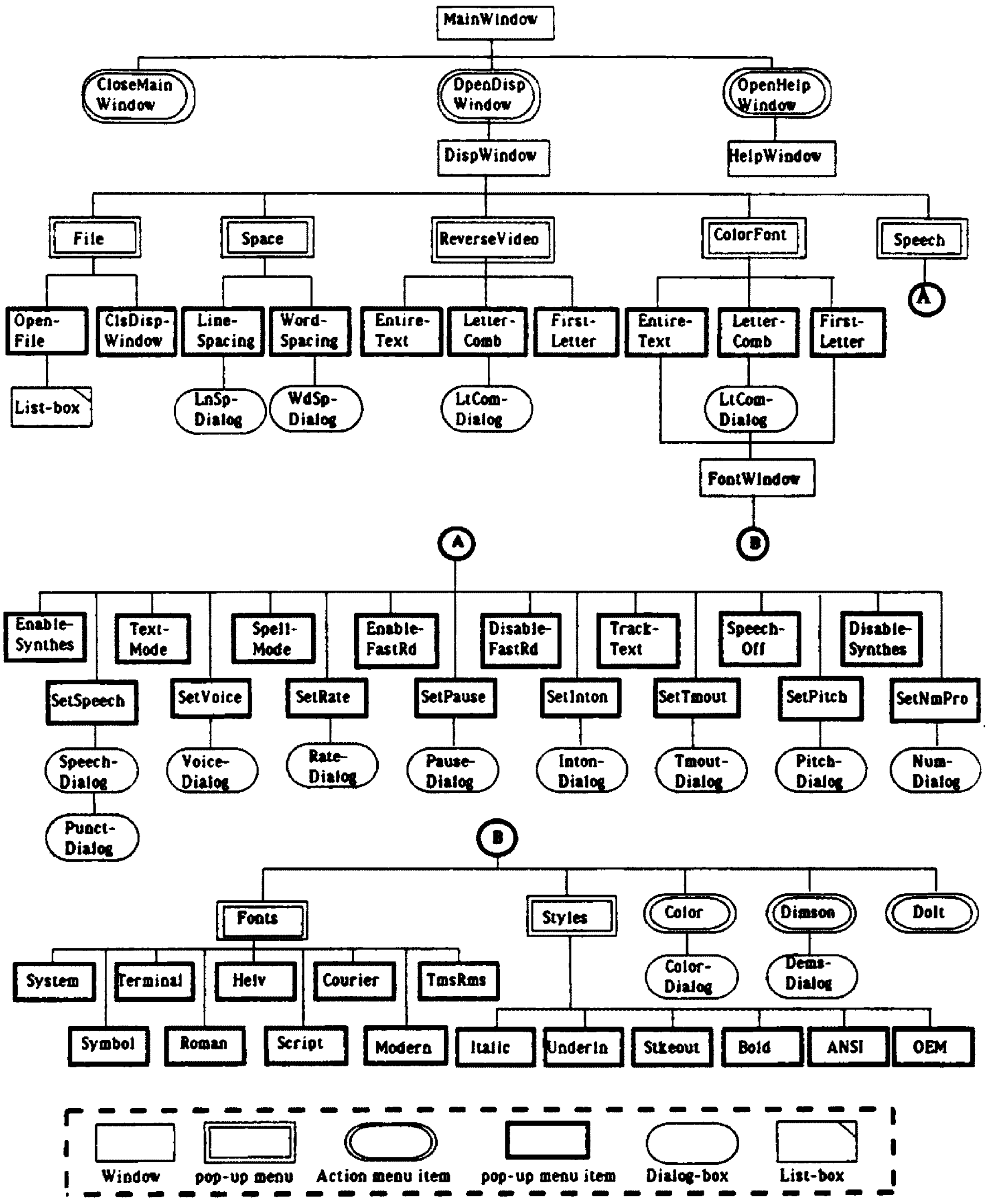


Figure 3.9 User-Interface

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter Four

Implementation

4.1. Chapter Overview

Although object-oriented programming has been around for about 20 years, interest in it and usage of it have increased tremendously in the last 5 years. OOP's recent popularity is the result of several factors. The most significant factor is that "OOP improves code reuse by using less complex, loosely coupled, highly cohesive components" [Duff and Howard 1990]. OOP technology is a real breakthrough in programming since building highly reusable software components is a difficult undertaking. This chapter focuses on the discussion of the implementation with Actor, an OOP language, of the display software of this project. The chapter starts with a brief introduction of MS-Windows, Actor, and their relationship. The chapter then provides an example to further describe the programming approach with the OOP technique. The end of the chapter discussed code flexibility and reusability of this project.

4.2. MS-Windows and Actor

MS-Windows is a graphical user-interface for PCs running under MS-DOS and PC-DOS. It provides an easy and friendly environment to the end-user. Actor is an object-oriented programming language which runs under MS-Windows and is used to produce windows applications.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.2.1. MS-Windows

Macintosh users have always had the advantage of a windowing environment and an easy-to-use interface, while early MS-DOS based PC users were at a disadvantage as they worked only in a non-windowing environment. However, today MS-Windows is well on its way to becoming the standard windowing environment for PCs. Among systems that run programs in a windowed or graphical environment on MS-DOS based PC's, MS-Windows has taken the lead in the past years.

4.2.2. Actor

Actor is both an object-oriented programming language and a sophisticated environment for developing MS-Windows applications on PCs. It works with Windows to provide a versatile environment for programmers as well as a simple, standardized interface for users. Actor is one of the four (Digitalk's Smalltalk/V and Smalltalk/V286, Interactive Software Engineering's Eiffel, Whitewater Group's Actor) widely used pure OOP languages. Pure and hybrid are two types of OOP languages. A pure language operates strictly within the rules of object-oriented programming. In other words, in a pure object-oriented language, all variables can be considered to be objects. A hybrid language (such as C+ +), on the other hand, is a language that is an object-oriented extension to a non-object-oriented language.

4.2.3. Relationship Between Actor and MS-Windows

Windows has the reputation of being easy for users but difficult for programmers. It is very good for the end-user who desires to use graphics-oriented programs

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

with a simple and standard interface. The programmer, however, has to learn a vast set of new functions and a new style of programming. Actor is designed especially for MS-Windows programming. Actor operates between MS-Windows and the programmer to simplify windows programming. Actor plus MS-Windows not only gives the user standardized applications but also provides the programmer with the capability to develop standard windows applications in a powerful and easy-to-use object-oriented environment.

Windows is an event-driven system, which means that programs respond to events that the user or other programs initiate [Urlocker 1989]. Events refers to actions such as clicking the mouse, pressing a key, or selecting a menu item. Windows operates on a message-passing paradigm. In other words, Windows sends a message to notify the program once an event occurs. For example, the corresponding window object receives a WM_LBUTTONDOWN message whenever the user clicks in a window with the left mouse button. The "WM" is mnemonic for a Windows Message. Actor has predefined window classes which behave in a standard way. In fact, a window is an instance of a window class in Actor. There are a wide array of predefined window classes (such as TextWindow, EditWindow ...) in Actor's class library. Once a window class is defined, a window of that class can be created with just one line of Actor code. To program with Actor, the programmer can define more specialized window classes as descendents of the standard window classes.

Actor provides simplified access to perhaps 85 percent of the Windows functions, while the remaining functions must be invoked using conventional programming methods [Cummings 1987]. This simplification makes creating a Windows application substantially easier and distinguishes Actor from other OOP

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

languages. For instance, using the C programming language to create a window which displays "Hello world!" requires over a hundred lines of code. But just two lines of Actor code will be sufficient to achieve the same result. In addition, Actor handles memory management automatically, so that the programmer does not need to worry about it. The following is a case study of employing polymorphism, inheritance and encapsulation for this project with Actor.

4.3. Case Study

One of this project's major tasks is to enable the user to modify the textual display attributes. Of all the display attributes, color and font receive the most modifications. The program uses a font window as a common interface to receive newly specified colors and fonts from the user. As it is stated in chapter three, three classes (TextAttributes, LetterCombAttributes and FirstLetterAttributes) are defined to handle text-attributes, letter-combination-attributes and first-letter-attributes respectively. Moreover, these three classes are sub-classes of the VisualAttributes class. Since objects of these three classes all use the same font window to get the user's requirements, the interface between these three classes and the FontWindow class becomes problematic in the implementation phase. In other words, it is very important that the way a font window object returns display attributes to different textual attributes objects. A language like C requires that each function have a unique name so that the function can be located at the compile time. Thus, different objects can only receive display attributes through different routines. However, with polymorphism, this can be done easily. The following code is part of the *command* method that is defined in the FontWindow class and is used to return display attributes to different objects:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
Def command(self, wP, lP)
{select
        case    wP = = IDM_ITALIC
        is      ......
        endCase
        case    wP = = IDM_CHARCOLOR
        is      ......
        endCase
        ......
        case    wP = = IDM_DOIT
        is      setUnderline(attribObj , underline);
                setItalic(attribObj , italic);
                setWeight(attribObj , weight);
                setFontColor(attribObj , fontColor);
                setBkColor(attribObj , bkColor);
                setHeight(attribObj , height);
                setWidth(attribObj , width);
                setCharSet(attribObj , charSet);
                setStrikeout(attribObj , strikeout);
                setFaceName(attribObj , fontList[fontId]);
                applyNewFonts(attribObj );
        endCase;
endSelect;
......
}
```

where: attribObj is an instance variable of the FontWindow class that

refers to a visual attributes object

underline, italic, weight, fontColor, ... are also instance variables of the

FontWindow class

set-"attributes"() are methods defined in the VisualAttributes class

applyNewFonts() is a method defined in the TextAttributes,

LetterCombAttributes and FirstLetterAttributes classes

This is the command-dispatcher of font window. It handles the font window's menu events. Each menu item corresponds to a "case" statement in the above code. For instance, if the user selects "Italic" from the menu, then the methods under "case wP = = IDM_ITALIC" will be invoked. Methods under "case wP = = IDM_DOIT" will be invoked if the user selects "DoIt" on the menu, so that the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

newly specified fonts and colors are applied to the display window.

4.3.1. Polymorphism

As stated in previous chapters, polymorphism refers to use the same name for different types of objects. This ability is based on the technique called "dynamic binding." The process of matching up a message with a method is called *binding* . Actor binds messages with methods at the run time, so it utilizes dynamic binding. Return to the code given above, message "applyNewFonts" is sent to *attribObj* which refers to different objects (textAttri, letterCombAttri, and firstLetterAttri) as the program runs. This is where polymorphism is employed. Figure 4.1 shows this polymorphism.

With this polymorphic implementation, *fontWindow* sends the message "applyNewFonts" regardless of the **applyNewFonts()** operation. Therefore, the relationship between *fontWindow* and objects (textAttri, letterCombAttri, and firstLetterAttri) is clear and stable. If two additional classes: SentenceAttributes and ParagraphAttributes were added into the system, then *fontWindow* would communicate with the corresponding objects (paragAttri and sentsAttri) through the same code given above.

4.3.2. Inheritance

Inheritance means that an object inherits all the attributes and methods from its parents. In the above code, *fontWindow* sends set-"attributes" messages to different objects (textAttri, letterCombAttri and firstLetterAttri). However, the corresponding methods are not defined in any of these objects' classes, but in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

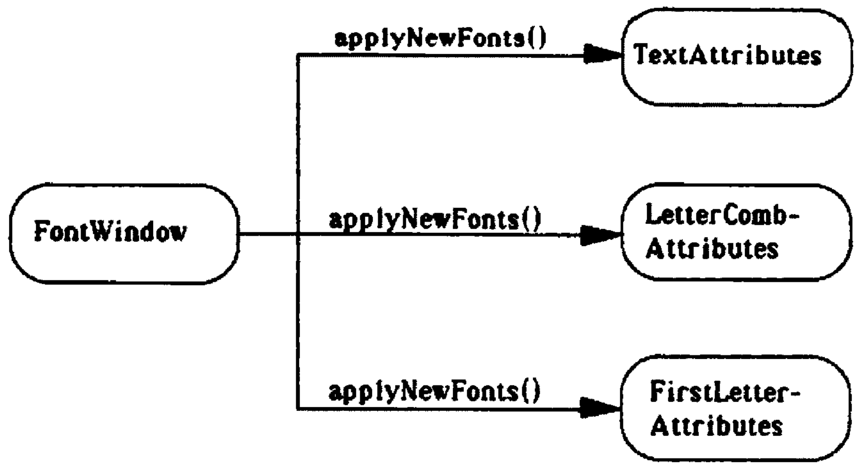


Figure 4.1 Example of polymorphism

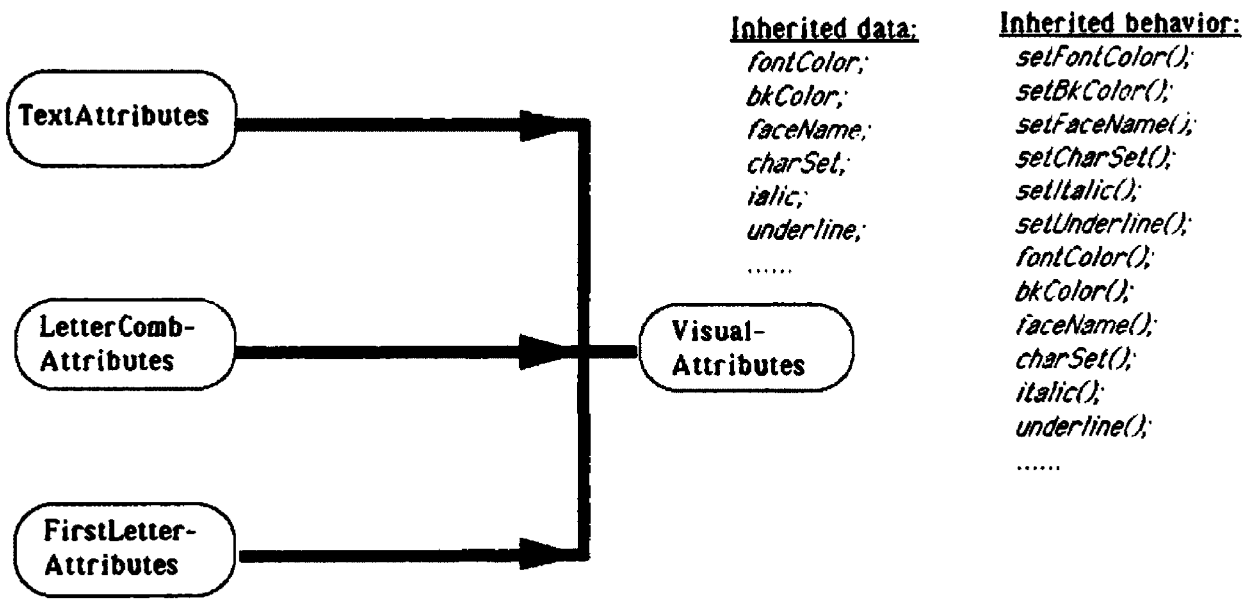


Figure 4.2 Example of inheritance

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

their parent class (VisualAttributes). The inheritance enables these objects inherit both data and methods from their parent, so that they can respond messages without redefining methods in their classes. Figure 4.2 depicts this part of the inheritance hierarchy.

4.3.3. Encapsulation

Encapsulation is the technical term for "information hiding." It is used to describe an object's protection of its private data from its clients (objects that use the resources of other objects). Actor provides two techniques (using messages and using "dot notation") for clients accessing an object's instance variables. Using messages follows the principle of encapsulation whereas "Dot notation" violates encapsulation. Moreover, *using messages* implies that an object's clients can not access its instance variables unless the object provides a public message that gives the access to its data in an abstract way. *"Dot notation"* allows clients to directly access an object's instance variables. Therefore, using messages is highly recommended and "dot notation" is strongly discouraged in Actor language usage.

As it shows in the above code, messages set-"attributes" are sent to different objects to pass on textual display attributes. For example, message setUnderline(attribObj, underline) is sent to *attribObj* for passing a newly specified underline attribute. In the display program, "message sending" is the only outside access to an object's instance variables. However, instead of using message sending to set *attribObj's* instance variables, this can be accomplished by: *attribObj.vUnderline := underline* , where *vUnderline* is an instance variable of *attribObj's* class. This direct-accessing *vUnderline* results in building a tight

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

coupling between *fontWindow* and *attribObj* , which means that modifications to the *attribObj's* class may effect *fontWindow's* class.

4.4. Flexibility and Reusability

Flexibility and reusability measure the quality of a software package. A highly flexible and reusable code makes a software package easily maintainable, so that the software package lives longer. Consider the display program: suppose that someone wants to use another kind of speech synthesizer, rather than ACCENT-SA. If the new synthesizer enables the same speech attributes, then the only modification would be made within the Synthesizer class; nothing would need to be changed in other classes. If another user wants to distinguish the last letter of each word, the program could be extended to handle this feature easily. This can be accomplished by creating another class: LastLetterAttributes (descendant of the VisualAttributes class) in the class hierarchy and adding no more than five short methods to the class. It is also necessary to add another method in the DispCollection class in order to traverse the last letter of each word in the entire text. Finally, corresponding menu items, and the command dispatcher of the DispWindow class need to be changed accordingly. Yet the whole structure does not change for this purpose, and 95 percent of the code will not be affected.

One might think that using an OOP language obviously signifies a better design. That is not necessarily true. To solve a certain problem, one might make one or two huge classes, or divide into many small ones. The OOD technique encourages small classes with loose coupling and high cohesion. The development of this project is an example. When I first started this project, I did

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

not really understand the substance of OOP except some terms such as object, class and method. I designed this project making dispWindow object responsibility for both window events and textual display attributes. As a result, the DispWindow class became a huge class and difficult to maintain. It is obviously not a good design from the OOP point of view, even though it was based on an object-oriented programming language. The current design is better since the dispWindow object no longer has the responsibility for textual display attributes. However, if I was to do the project over again, I would break down the DispWindow class even more, so as to create a class to handle all the key-actions for the display window. This experience shows that the quality of software does not merely depend on using an object-oriented programming language. Rather, OOP languages add features that allow implementation of well-factored, minimally coupled systems. Yet the programmer still has considerable leeway to build up different object models for a particular problem. Different object models can lead to various class hierarchies with different data-structures and methods design. Assuming that the problem has been decomposed properly, the utilization of polymorphism and encapsulation still affect the quality of the software. It is true that OOP provides good preconditions for more flexible and reusable software, but it is still a considerable challenge for the programmer.

For a programmer who does not know OOP techniques and languages well, the start-up time expenditure is quite considerable. OOD requires the knowledge of a particular object-oriented programming language. OOP languages can not be mastered in a short time. They are quite different from traditional programming languages. The time and efforts are fairly considerable in order to learn the OOP language, along with the OOD technique. As a result, I spent at least 50 percent of the development time learning the language before getting into the formal

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

coding.

However, once I became familiar with the Actor language and better understood the concept of OOP, to improve the program was easy. For instance, I spent less than a month for the program improvement which includes design and coding. This shows that object-oriented languages do simplify the work for the programmer and Actor is a good language for developing windows application.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter Five

Conclusion

5.1. What the Software Accomplished

This software project's idea originates from Horn's experimental results [Horn 1989]. This project's primary goal is to create a program that provides a dynamic and changeable textual display environment for the dyslexic reader. It focuses on carrying the aforementioned idea toward a more practical reality rather than simply producing a software product. In addition, this software development process gave the author a chance to try out both object-oriented design, graphic user-interface programming, and the Actor programming language. Thus, it is more an practice than an experiment. Though the program has not yet been used by dyslexics, it does achieve the requirements specified in chapter two. Therefore, the author believes the program to be not only a good start toward textual clarity but also a useful tool in the future for dyslexic readers.

5.2. Future Enhancements

This display software's future enhancement will require attention to two aspects: extension of the application functions and refinement in terms of software design and implementation techniques.

The extension of the application functions depends upon the dyslexic user's requirements. As this paper states in chapter two, this software is a start-up

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

program. It contains basic textual display functions which are aimed at the dyslexic's common problems. In fact, dyslexic individuals have various reading difficulties; many of these problems may not be considered in this program. However, this program is flexible for future functional extension. For example, the dyslexic may like to use different display attributes for the last letter of each word, a particular paragraph, a certain line and so on. Since the program applies the object-oriented design technique, the objects have loose coupling relationships. This allows the program to achieve the above requirements with fewer side affects for existing functions.

From the viewpoint of software design and implementation technique, several refinements could be considered. The first is the user-interface for font input. The current software uses a font window as an interface to communicate user's fonts updating. The alternative is to add font options on the Text Display Window's main menu, thus simplifying the font-selection process and making it more convenient for the user. Another consideration is to use a more direct method for color input. The current user-interface for color modification is the numerical RGB color input dialogue, which requires the user to input three numbers in order to balance the RGB and to obtain the desired color. By using RGB color bars and moving the mouse to adjust color bar scrolls, the user can visually calibrate his/her desired textual display. Finally, it maybe helpful to provide two versions of text format in terms of display manner: normal display, where text extends beyond the display window's width, and a wraparound display, where text stays within the display window's boundaries. The current program provides text display without special handling for text which extends beyond the side borders of the window, so that the user must use a mouse or space-bar to trace the text when the line length exceeds the window's width. A wraparound text display would give the user

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

another version of the text with automatic wraparound line handling that would eliminate the user's needs to trace text beyond the window, thereby enhancing the user's reading speed. Until further experimentation allows dyslexics to suggest further enhancements, these are just a few refinements the author can now recommend.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix A

Class --- Behaviors --- Collaborators

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CLASS-NAME	BEHAVIORS	COLLABORATORS
DispCollection	a collection of lines; handle different searches on the input file	DispWindow
DisplayApp	start-up the program and create the main window	MainWindow
DispWindow	handle the menu events, and draw text file on the display window	MainWindow TextAttributes LetterCombAttributes FirstLetterAttributes Synthesizer DispCollection project.h @
FirstLetterAttributes	handle first letter of words' display attributes	DispWindow FontWindow VisualAttributes
FontWindow	behave like a text window; provide font and color menu; handle menu events	DispWindow TextAttributes LetterCombAttributes FirstLetterAttributes
HelpWindow	bring up help messages as desired	MainWindow *.hlp (&)
IntonDialog	show a sentence-level-intonation selection dialog; get input from users	Synthesizer intonati.h @
LetterCombAttributes	handle letter-combination display attributes	DispWindow FontWindow
MainWindow	parent window of the display window and the help window	DisplayApp DispWindow HelpWindow main.h @
NumberDialog	show a number-processor selection dialog; get input from users	Synthesizer number.h @
PauseDialog	show a space-pause selection dialog; get input from users	Synthesizer pause.h @
PitchDialog	show a pitch selection dialog; get input from users	Synthesizer pitch.h @
PunctDialog	show a punctuation dialog; get input from users	Synthesizer punctuation.h @
RateDialog	show a speech-rate selection dialog; get input from users	Synthesizer rate.h @
SpeechDialog	show a speech dialog; get input from users	Synthesizer speech.h @

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Synthesizer	connect to the serial port,; handle the speech options	DispWindow SpeechDialog VoiceDialog PitchDialog PunctDialog RateDialog IntonDialog NumberDialog PauseDialog TimeoutDialog
TextAttributes	handle line space, word space	DispWindow FontWindow VisualAttributes
TimeoutDialog	show a time-out selection dialog; get input from users	Synthesizer timeout.h @
VisualAttributes	parent class of TextAttributes, FirstLetterAttributes and LetterCombAttributes; handle common textual display attributes	TextAttributes LetterCombAttributes FirstLetterAttributes
VoiceDialog	show a voice selection dialog; get input from users	Synthesizer voice.h @

@ --- refers to the resources (.h) files

& --- refers to set of (.hlp) files which provide help messages

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix B

Method-Description

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

METHOD-NAME	FROM	INPUT	RESULTS
alterMenu	FontWindow	value menuItem checked unchecked	change the menu item and return the new value
applyNewFonts	TextAttributes		apply new fonts for the entire text to the display window
applyNewFonts	FirstLetterAttributes		apply new fonts for the first letter of each word to the display window
applyNewFonts	LetterCombAttributes		fill in the dictionary then apply new font of letter-combination to the display window
applySpeech	Synthesizer	str	send str to the serial port
arrowDown	DispWindow	adj	adjust window's view when "arrow-down" key is used
arrowLeft	DispWindow	adj	adjust window's view when "<-" key is used
arrowRight	DispWindow	adj	adjust window's view when "->" key is used
arrowUp	DispWindow		adjust window's view when "arrow-up" key is used
changeFont	FontWindow	idVal	change new fonts to the font window
charIn	DispWindow	wP lP	turn off the display window's character input ability
charIn	HelpWindow	wP lP	turn off the help window's character input ability
closeSerialPort	Synthesizer		disable the speech synthesizer and close the serial port
command	MainWindow	wP lP	menu events handling for the main window
command	DispWindow	wP lP	menu events handling for the display window
command	HelpWindow	wP lP	menu events handling for the help window

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

command	FontWindow	wP lP	menu events handling for the font window; bring back the new fonts to the display window through attribObj
createDictionary	LetterCombAttributes		set letterCombDict (#)
createMenus	FontWindow		create menu for the font window
dictionary	LetterCombAttributes		return letterCombDict
disableFastRead	Synthesizer		send signal to synthesizer to disable the fast read
drawLetter	DispWindow	textual attributes (%)	draw letter with passed-in attributes
drawText	DispWindow	hdc textual attributes (%)	draw the entire text with passed-in attributes
enableFastRead	Synthesizer		send signal to synthesizer to enable the fast read ability
enableSpeechMenu	DispWindow		make speech menu available for selecting
exchangeColors	TextAttributes		swap vFontColor (#) and vBkColor (#)
findDown	DispCollection	strtLine strtChar lSp	return a tuple of (x, y, word). x and y are location of word which is down to the current char
findFirst	DispCollection	strtLine strtChar	return a tuple: (x, y, ch). x and y are starting pos of first char
findLeft	DispCollection	strtLine strtChar lSp	return a tuple of (x, y, word). x and y are location of word which is left to the current char
findRight	DispCollection	strtLine strtChar lSp	return a tuple of (x, y, word). x and y are location of word which is right to the current char
findString	DispCollection	str strtLine strtChar	return the position (x, y) of str

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

findUp	DispCollection	strtLine strtChar lSp	return a tuple of (x, y, word). x and y are location of word which is up to the current char
findWords	DispCollection	strtLine strtChar wSp	return a tuple: (x, y, str) of each word. x and y are starting pos of word
flushSpeakBuffer	Synthesizer		stop synthesizer working immediately
getChar	DispCollection	strtLin strtLoc	return "true" if the char in given location is a sign or punctuation
getCurrentChar	DispCollection	line char	return "true" if current char is neither "space" nor "nil"
getDimensions	FontWindow		create dynamic dialog and get: height (#) and width (#) from users
getFonts	FirstLetterAttributes		open the font window to get attributes for first letter of each words
getFonts	LetterCombAttributes		open the font window to get attributes for letter-combinations
getFonts	TextAttributes		open the font window to get attributes for the entire text
getFontList	FontWindow		enumerate over the fonts available to the font system and return a font list
getLastChar	DispCollection	lSp	return the index of the last char in the entire text
getLeftFirst	DispCollection	ln	return the index of left-first char in the given line

getNextPos	DispCollection	endL
		endC
			return location (x, y) of next position while cursor moves "down"

getPreChar	DispCollection	endL
		endC
			return the pre-char's index number

getPreLine	DispCollection	endL	return the pre-line's index number

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

getPrePos	DispCollection	endL endC	return location (x, y) of pre-position while cursor moves "up"
getRGB	FontWindow		create dynamic dialog and get RGB color from users
getRight	DispCollection	ln	return the size of given line
gotFocus	FontWindow	hWndPrev	redefine gotFocus to do nothing
graySpeechMenu	DispWindow		disconnect speech menu
handleColorFont	DispWindow	fontColor	dispatch the users' choice for color & font changing
handleFileOpen	DispWindow		open a file and enable the textual display menu
handleReverseVideo	DispWindow	choice	dispatch the user's choice for reverse-video
handleSpace	DispWindow	space	dispatch space selection events and reset caretExt (#)
init	DispWindow		set: speech (#), caretExt (#) & initialize the display window
init	DisplayApp	command-Str	bring up the main window
init	FontWindow		set fonts for the font window
init	Synthesizer		set speech attributes (@)
init	VisualAttributes		set textual attributes (%)
initAttributes	DispWindow		set: textAttri (#), letterCombAttri (#), firstAttri (#) and send message to letterCombAttri and create letter-comb-dictionary
initTextColors	FontColor	hdc	select font attributes and reset char color and bk color
initTextMetrics	DispWindow		set default height & width for letters
initWorkText	DispWindow		set workText (#)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

insertText	DispCollection	aStr line pos lSp	insert a string of lines into the collection at specified line and char position
keyDown	DispWindow		reverse video for the word that is below the current word
keyLeft	DispWindow		reverse video for the word that is left of the current word
keyRight	DispWindow		reverse video for the word that is to the right of the current word
keyUp	DispWindow		reverse video for the word that is above the current word
lastChar	DispWindow		return the last char of the whole text
leftFirstChar	DispWindow	line	return the index of the first char of the given line
lineSpace	TextAttributes		return vLineSpaceL (#)
loadFile	DispWindow		load a file to the display window
maxFileSize	DispWindow		return the maximum file size available to read
maxFileSize	HelpWindow		return the maximum file size available to read
newFont	FontWindow	idVal	change the font for strings on the font window
openFile	DispWindow	fl	open a new file & replace work text with its contents
openFile	HelpWindow	fl	open a new file & replace work text with its contents
openSerialPort	Synthesizer		init and set-up a serial port
outRange	DispWindow	strtLine strtChar	return "true" if the current position is out of right-bottom corner of the window
paint	DispWindow	hdc	repaint the window

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

paintFirstLetter	DispWindow		get first-letter attributes from firstAttri then send message "drawLetter"
paintLetterComb	DispWindow		get letter-comb attributes from letterCombAttri then send message "drawLetter"
paintText	DispWindow	hdc	get text attributes from textAttri, then send message "drawText"
readText	DispWindow	fName	replace the selection range with the contents of a file
readText	HelpWindow	fName	replace the selection range with the contents of a file
reverseCombStatus	LetterCombAttributes		return reverseComb (#)
reverseFirstLetter	DispWindow		reverse video for the first letter of each word
reverseFirstStatus	FirstLetterAttributes		return reverseFirst (#)
reverseLetterComb	DispWindow		reverse video for the letter combination
reverseText	DispWindow		send message "exchangeColors" to textAttri
rightChar	DispWindow	line	return the index of the last char of given line in text
selectFont	FontWindow	hdc	select the new fonts
setArray	DispCollection		init numBlank (#)
setAttribObj	FontWindow	txtAttrib	set attribObj (#)
setBkColor	FontWindow	newBkColor	set bkColor (#)
setCharColor	FontWindow	newFontColor	set fontColor (#)
setLineSpace	TextAttributes		create an input dialog to get user's desired line-space; set vLineSpaceL (#) & vLineSpaceP (#)
setReverseComb	LetterCombAttributes		set reverseComb (#)
setReverseFirst	FirstLetterAttributes		set reverseFirst (#)
setSpellMode	Synthesizer		send signal to synthesizer to change to spell mode

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

setTextMode	Synthesizer		send signal to synthesizer to change to speech mode
setWordSpace	TextAttributes		create an input dialog and get user's desired word space: vWordSpace(#)
show	FontWindow	state	show the font window with two strings on it
speechOn	Synthesizer		return "true" if speechFlag is not nil
trackText	DispWindow		reverse video on each word when tracing the entire text (used while speaking)
transSpeech	Synthesizer	wp	translate dialog message to a string of chars that numerate speech options
transSpeechII	Synthesizer	wp	translate dialog message to a string of chars that numerate speech options
updateAttributes	FirstLetterAttributes		return "true" if attriChangeFlag < > nil
updateAttributes	LetterCombAttributes		return "true" if letterCombDict < > nil
visChars	DispWindow		return the number of chars which are visible in the current window
visibleChar	DispWindow	strtLine strtChar	return "true" if the current position is visible
WM_KEYUP	DispWindow	wP IP	respond to the arrow keys
wordLimits	DispCollection	line char	return the position in (x,y) of start and end the current word
wordSpace	TextAttributes		return vWordSpace (#)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

"%" --- refers to instance variables for textual-attributes

"@" --- refers to instance variables for speech attributes

"#" --- refers to corresponding instance variables

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix C

Data-Dictionary

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

NAME	TYPE	FROM	USED-BY
alterMenu	method	FontWindow	FontWindow
applyNewFonts	method	TextAttributes LetterCombAttributes FirstLetterAttributes	FontWindow
applySpeech	method	Synthesizer	DispWindow
arrowDown	method	DispWindow	DispWindow
arrowLeft	method	DispWindow	DispWindow
arrowRight	method	DispWindow	DispWindow
arrowUp	method	DispWindow	DispWindow
attribObj	instance	FontWindow	TextAttributes LetterCombAttributes FirstLetterAttributes
attriChangeFlag	instance	FirstLetterAttributes	FirstLetterAttributes
bkColor	instance	FontWindow	FontWindow
caretExt	instance	DispWindow	DispWindow
changeFont	method	FontWindow	FontWindow
charIn	method	DispWindow	DispWindow
charIn	method	HelpWindow	HelpWindow
charSet	instance	FontWindow	FontWindow
closeSerialPort	method	Synthesizer	DispWindow
command	method	Dialog classes @	Dialog classes @
command	method	DispWindow	DispWindow
command	method	FontWindow	FontWindow
command	method	HelpWindow	HelpWindow
command	method	MainWindow	MainWindow
createDictionary	method	LetterCombAttributes	DispWindow LetterCombAttributes
createMenus	method	FontWindow	FontWindow
dictionary	method	LetterCombAttributes	DispWindow
disableFastRead	method	Synthesizer	DispWindow
DispCollection	class	OrderedCollection	DispWindow
DisplayApp	class	Object	*the user*
DispWindow	class	EditWindow	MainWindow
drawLetter	method	DispWindow	DispWindow
drawText	method	DispWindow	DispWindow
enableFastRead	method	Synthesizer	DispWindow
enableSpeechMenu	method	DispWindow	DispWindow
exchangeColors	method	TextAttributes	DispWindow
fileName	instance	DispWindow	DispWindow
findDown	method	DispCollection	DispWindow
findFirst	method	DispCollection	DispWindow
findLeft	method	DispCollection	DispWindow
findRight	method	DispCollection	DispWindow
findString	method	DispCollection	DispWindow
findUp	method	DispCollection	DispWindow
findWords	method	DispCollection	DispWindow
firstAttri	instance	DispWindow	DispWindow

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

FirstLetterAttributes	class	VisualAttributes	DispWindow FontWindow
flipFormat	method	Dialog classes @	Dialog classes @
flushSpeakBuffer	method	Synthesizer	DispWindow
fontColor	instance	FontWindow	FontWindow
fontId	instance	FontWindow	FontWindow
fontList	instance	FontWindow	FontWindow
FontWindow	class	TextWindow	TextAttributes LetterCombAttributes FirstLetterAttributes
fontWindow	instance	VisualAttributes	TextAttributes LetterCombAttributes FirstLetterAttributes
getChar	method	DispCollection	DispCollection
getCurrentChar	method	DispCollection	DispWindow
getDimensions	method	FontWindow	FontWindow
getFontList	method	FontWindow	FontWindow
getFonts	method	TextAttributes LetterCombAttributes FirstLetterAttributes	DispWndow
getLastChar	method	DispCollection	DispWindow
getLeftFirst	method	DispCollection	DispCollection DispWindow
getNextPos	method	DispCollection	DispWindow
getPreChar	method	DispCollection	DispWindow
getPreLine	method	DispCollection	DispWindow
getPrePos	method	DispCollection	DispWindow
getRGB	method	FontWindow	FontWindow
getRight	method	DispCollection	DispWindow
gotFocus	method	FontWindow	FontWindow
graySpeechMenu	method	DispWindow	DispWindow
handleColorFont	method	DispWindow	DispWindow
handleFileOpen	method	DispWindow	DispWindow
handleReverseVideo	method	DispWindow	DispWindow
handleSpace	method	DispWindow	DispWindow
height	instance	FontWindow	FontWindow
HelpWindow	class	EditWindow	MainWindow
init	method	DisplayApp	DisplayApp
init	method	DispWindow	DispWindow
init	method	FontWindow	FontWindow
init	method	Synthesizer	Synthesizer
init	method	VisualAttributes	VisualAttributes
initAttributes	method	DispWindow	DispWindow
initDialog	method	Dialog classes @	Dialog classes @
initTextColors	method	FontWindow	FontWindow
initTextMetrics	method	DispWindow	DispWindow
initWorkText	method	DispWindow	DispWindow

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

insertText	method	DispCollection	DispWindow HelpWindow
intonation	instance	IntonDialog	IntonDialog Synthesizer
IntonDialog	class	InputDialog	Synthesizer
italic	instance	FontWindow	FontWindow
keyDown	method	DispWindow	DispWindow
keyLeft	method	DispWindow	DispWindow
keyRight	method	DispWindow	DispWindow
keyUp	method	DispWindow	DispWindow
lastChar	method	DispWindow	DispWindow
leftFirstChar	method	DispWindow	DispWindow
letterCombAttri	instance	DispWindow	DispWindow
LetterCombAttributes	class	VisualAttributes	DispWindow FontWindow
letterCombDict	instance	LetterCombAttributes	LetterCombAttributes DispWindow
lineSpace	method	TextAttributes	DispWindow
loadFile	method	DispWindow	DispWindow
MainWindow	class	Window	DisplayApp
maxFileSize	method	DispWindow	DispWindow
maxFileSize	method	HelpWindow	HelpWindow
newFont	method	FontWindow	FontWindow
number	instance	NumberDialog	NumberDialog Synthesizer
numberBlank	instance	DispCollection	DispCollection
NumberDialog	class	InputDialog	Synthesizer
openFile	method	DispWindow	DispWindow
openFile	method	HelpWindow	HelpWindow
openSerialPort	method	Synthesizer	DispWindow
outRange	method	DispWindow	DispWindow
paint	method	DispWindow	DispWindow
paintFirstLetter	method	DispWindow	DispWindow
paintLetterComb	method	DispWindow	DispWindow
paintText	method	DispWindow	DispWindow
parWind	instance	VisualAttributes	FirstLetterAttributes LetterCombAttributes TextAttributes
pause	instance	PauseDialog	PauseDialog Synthesizer
PauseDialog	class	InputDialog	Synthesizer
pitch	instance	PitchDialog	PitchDialog Synthesizer
PitchDialog	class	InputDialog	Synthesizer
PunctDialog	class	InputDialog	Synthesizer
punctuation	instance	PunctDialog	PunctDialog Synthesizer

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

rate	instance	RateDialog	RateDialog Synthesizer
RateDialog	class	InputDialog	Synthesizer
readText	method	DispWindow	DispWindow
readText	method	HelpWindow	HelpWindow
reverseComb	instance	LetterCombAttributes	LetterCombAttributes
reverseCombStatus	method	LetterCombAttributes	DispWindow
reverseFirst	instance	FirstLetterAttributes	FirstLetterAttributes
reverseFirstLetter	method	DispWindow	DispWindow
reverseFirstStatus	method	FirstLetterAttributes	DispWindow
reverseLetterComb	method	DispWindow	DispWindow
reverseText	method	DispWindow	DispWindow
rightChar	method	DispWindow	DispWindow
sAbbrevation (&)	instance	Synthesizer	Synthesizer
sDash (&)	instance	Synthesizer	Synthesizer
selectFont	method	FontWindow	FontWindow
serialPort	instance	Synthesizer	Synthesizer
setArray	method	DispCollection	DispWindow
setAttribObj	method	FontWindow	TextAttributes LetterCombAttributes FirstLetterAttributes
setBkColor	method	FontWindow	FontWindow
setCharColor	method	FontWindow	FontWindow
setLineSpace	method	TextAttributes	DispWindow
setParent	method	VisualAttributes	DispWindow
setReverseComb	method	LetterCombAttributes	DispWindow
setReverseFirst	method	FirstLetterAttributes	DispWindow
setSpellMode	method	Synthesizer	DispWindow
setTextMode	method	Synthesizer	DispWndow
setWordSpace	method	TextAttributes	DispWindow
show	method	FontWindow	TextAttributes LetterCombAttributes FirstLetterAttributes
sIntonation (&)	instance	Synthesizer	Synthesizer
sNumber (&)	instance	Synthesizer	Synthesizer
sPause (&)	instance	Synthesizer	Synthesizer
speech	instance	DispWindow	DispWindow
SpeechDialog	class	InputDialog	Synthesizer
speechFlag	instance	Synthesizer	Synthesizer
speechOn	method	Synthesizer	DispWindow
speechAbbre	instance	SpeechDialog	SpeechDialog Synthesizer
speechDash	instance	SpeechDialog	SpeechDialog Synthesizer
speechPunct	instance	SpeechDialog	SpeechDialog Synthesizer
sPitch (&)	instance	Synthesizer	Synthesizer
sPunctSet (&)	instance	Synthesizer	Synthesizer

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

sPunctuation (&)	instance	Synthesizer	Synthesizer
sRate (&)	instance	Synthesizer	Synthesizer
sTimeout (&)	instance	Synthesizer	Synthesizer
strikeout	instance	FontWindow	FontWindow
sVoice (&)	instance	Synthesizer	Synthesizer
Synthesizer	class	Object	DispWindow
textAttri	instance	DispWindow	DispWindow
TextAttributes	class	VisualAttributes	DispWindow FontWindow
timeout	instance	TimeoutDialog	TimeoutDialog Synthesizer
TimeoutDialog	class	InputDialog	Synthesizer
trackText	method	DispWindow	DispWindow
trackText	instance	DispWindow	DispWindow
transSpeech	method	Synthesizer	Synthesizer
transSpeech	method	Synthesizer	Synthesizer
updateAttributes	method	FirstLetterAttributes	DispWindow
updateAttributes	method	LetterCombAttributes	DispWindow
vBkColor (#)	instance	VisualAttributes	VisualAttributes; DispWindow
vCharSet (#)	instance	VisualAttributes	VisualAttributes DispWindow
vFaceName (#)	instance	VisualAttributes	VisualAttributes DispWindow
vFontColor (#)	instance	VisualAttributes	VisualAttributes DispWindow
vHeight (#)	instance	VisualAttributes	VisualAttributes DispWindow
visChars	method	DispWindow	DispWindow
visibleChar	method	DispWindow	DispWindow
VisualAttributes	class	Object	DispWindow
vItalic (#)	instance	VisualAttributes	VisualAttributes DispWindow
vLineSpaceL	instance	VisualAttributes	TextAttributes DispWindow
vLineSpaceP	instance	VisualAttributes	TextAttributes DispWindow
voice	instance	VoiceDialog	VoiceDialog Synthesizer
VoiceDialog	class	InputDialog	Synthesizer
vStrikout (#)	instance	VisualAttributes	VisualAttributes DispWindow
vUnderline (#)	instance	VisualAttributes	VisualAttributes DispWindow
vWeight (#)	instance	VisualAttributes	VisualAttributes DispWindow
vWidth (#)	instance	VisualAttributes	VisualAttributes DispWindow

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

vWordSpace	instance	VisualAttributes	TextAttributes DispWindow
weight	instance	FontWindow	FontWindow
width	instance	FontWindow	FontWindow
WM_KEYUP	method	DispWindow	DispWindow
wordLimits	method	DispCollection	DispCollection DispWindow
wordSpace	method	TextAttributes	DispWindow

"@" -- refers to that same method is defined-in / used-by Dialog-classes

"&" -- refers to that corresponding "get" methods exist in Dialog-classes and "send" methods exist in methods exist in both Synthesizer and DispWindow classes

"#" -- refers to that corresponding "set" methods and *return* methods exist in the VisualAttributes class

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Bibliography

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ACCENT^{TM} USER'S MANUAL (VERSION 3.1). (1986-1989). San Jose, California: Aicom Corporation.

Actor User's Manual. (1990). Evanston, Illinois: The Whitewater Group.

Asymetrix Corp. (May 1990). Bellevue, Washington.

Beck, Kent, & Cunningham, Ward. (1989). A Laboratory For Teaching Object-Oriented Thinking. OOPSLA

Birtwistle, Graham, & Dahl, J. Ole, & Myhrhaug, Bjorn, & Nygaard, Kristen. (1973). Simula Begin. Philadelphia: Auerbach Press.

Bobrow, G. Daniel. (1989, May 1). The Object of Desire. DATAMATION.

Bond, George. (1990, July). Actor Sets a New Stage for OOP. BYTE.

Booch, Grady. (1991). Object Oriented Design with Applications. Redwood City, California: The Benjamin/Cummings Publishing Company, Inc.

Borland International. (1987 1988). Scotts Valley, California.

Coad, Peter, & Yourdon, Edward. (1990). Object-oriented Analysis. Englewood Cliffs, New Jersey: Prentice Hall Inc.

Cox, J. Brad. (1986). Object Oriented Programming: An Evolutionary Approach. Sandy Hook, Connecticut: Addison-Wesley Publishing Company.

Cummings, Stephen. (1987, October 27). Actor Gives Windows Programmers a Hand. PC WEEK.

Demarco, Tom. (1979, 1978). Structured Analysis and System Specification. Englewood Cliffs, New Jersey: Yourdon Press, A Prentice-Hall Company.

Dodani, H. Mahesh, & Hughes, E. Charlesand, & J. Michael Moshell. (1989, March). Separation of Powers. BYTE

Doler, Kathleen. (1989, September 18). OOPS, an Old Friend Aids Language Developers. PC WEEK.

Duff, Chuck, & Howard, Bob. (1990, October). Migration Patterns: Moving to object-oriented technology is more involved than simply buying a compiler. BYTE.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Duntemann, Jeff. (1990, April/May). Our Object All Sublime. PC TECHNIQUES.

Horn, Christie. (1989). Reported in the American Higher Special Services Program Post-secondary Education (AHSSPPE) computer special interest group meeting, Seattle.

Hummel, L. Robert. (1989, May 30). Actor: Object-oriented Development Tool Lessens MS Windows Programming Load. PC MAGAZINE.

Jakupcak, Michael. [Personal Interview]. March, 1990-March, 1991.

Keough, Lee. (1989, February). The New Face of Computing. COMPUTER DECISIONS

Kerscher, George. [Personal Interview]. March, 1990-March, 1991.

Kim, Won, & Lochovsky, H. Frederick. (1989). Object-Oriented Concepts, Databases, and Applications. New York, New York: a division of the Association for Computing Machinery, Inc.

Knowledge Garden Inc. (May 1990). Nassau, New York.

Lazzaro, J. Joseph. (1990, August). Opening Doors For The Disabled. BYTE.

Martin, James. (1989, September 4). OOP Holds Promise of Simplifying Computer Programming. PC WEEK.

Martin, James. (1989, September 18). OOP's Intuitive Interface Widens Range of Applications. PC WEEK.

Meyer, Bertrand. (1988). Object-oriented Software Construction. New Jersey: Prentice Hall Inc.

Micallef, Josephine. (1988, April/May). Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages. JOOP.

Microsoft Corp. (1990). Redmond, Washington.

Microsoft Corp. (1985-1990). Redmond, Washington.

O'Connell, Daniel. (1989, September 18). Razing structured approaches. COMPUTERWORLD.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Petzold, Charles. (1988). Programming Windows. Redmond, Washington: Microsoft Press.

Rumbaugh, James, & Blaha, Michael, & Premerlani, William, & Eddy, Frederick, & Lorensen, William. (1991). Object-oriented Modeling and Design. Englewood Cliffs, New Jersey: Prentice Hall Inc.

Sherer, M. Paul. (1990, January 15). Actor 2.0 Breaks Through 640K-Byte Limit on Code. PC WEEK.

Terry, Chris. (1989, November 9). Object-Oriented Programming: Objects facilitate modular, reusable code. EDN.

Thomas, Dave. (1989, March). What's in an Object?. BYTE

Thompson, Tom. (1989, March). The Next Step. BYTE.

The Whitewater Group. (1990). Evanston, Illinois.

The Whitewater Group. For Faster, Better, Smaller, Smarter Windows Programming.

University of California. (1979). Berkeley, California.

Urlocker, Zack. (1989, November/December). Abstracting the User Interface. JOOP.

Urlocker, Zack. (1990, May). Object-Oriented Programming For Windows: Using OOP to develop applications for Microsoft Windows. BYTE.

Urlocker, Zack. (1989, March). Whitewater's Actor: An Introduction to Object-Oriented Programming Concepts. MICROSOFT SYSTEM JOURNAL.

Wegner, Peter. (1989, March). Learning the Language. BYTE

Xerox Corporation. (1983). Salt Lake City, Utah.

Yourdon, Edward, & Constantine, L. Larry. (1979). Structured Design: Fundamentals of a Displine of Computer Program and System Design. Englewood Cliffs, New Jersey: Prentice Hall Inc.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.