Visual Programming: Concepts and Implementations

Elizabeth Howard
Miami University, commons-admin@lib.muohio.edu

TECHNICAL REPORT: MU-SEAS-CSA-1994-000

Visual Programming:
Concepts and Implementations

Elizabeth Vera Howard

ABSTRACT

VISUAL PROGRAMMING:
CONCEPTS AND IMPLEMENTATIONS

by Elizabeth Vera Howard

The computing environment has changed dramatically since the advent of the computer. Enhanced computer graphics and sheer processing power have ushered in a new age of computing. User interfaces have advanced from simple line entry to powerful graphical interfaces. With these advances, computer languages are no longer forced to be sequentially and textually-based. A new programming paradigm has evolved to harness the power of today's computing environment - *visual programming*. Visual programming provides the user with visible models which reflect physical objects. By connecting these visible models to each other, an executable program is created. By removing the inherent abstractions of textual languages, visual programming could lead computing into a new era.

This paper will introduce the concepts of visual computing. A set of evaluative criteria for visual programming languages has been developed and will be used to compare two visual languages: National Instrument's LabVIEW and The Gunakara Sun Systems' ProGraph.

VISUAL PROGRAMMING: CONCEPTS AND IMPLEMENTATIONS

A Thesis

Submitted to the

Faculty of Miami University

in partial fulfillment of

the requirements for the degree of

Master of Systems Analysis

Department of Systems Analysis

by

Elizabeth Vera Howard

Miami University

Oxford, Ohio

1994

Advisor [illegible]

Reader [illegible]

Reader Valerie V. Cross

Reader Alton F. Sanders

Table of Contents

This thesis is dedicated to

Dr. Deb Schindler and Mr. Gerald Schindler for not only providing a roof over my head but also making Oxford my home.

And to Mrs. Faye Howard and Mr. Hager Howard for being wonderful friends and simply the world's best Mom and Dad.

Acknowledgments

I am especially indebted to Dr. James Kiper, whose experience and knowledge of visual programming, provided the basis for my thesis.

I would also like to extend a special thank-you to Ms. Chrissandra Griffin, who listened to my whining, whimpering, and complaining with great patience and understanding!

1.0 Introduction

Early programming languages mimicked the needs of von Neumann computer architecture hardware and necessarily used sequential, simple, character-based input and output statements. Programming languages progressed from machine and assembly code to natural language-based textual approaches in an effort to make algorithms more readable. Textual languages are inherently one-dimensional and focus on sequential execution of algorithms. This forced programmers essentially to restrict their designs to a linear organization. The facilities that textual programming languages "provide for describing algorithms correspond more closely to how computers operate than to the cognitive or perceptual processes of the programmer." [Cox *et al* 1989].

Computer hardware technology, however, has improved at an impressively high rate since the advent of the computer. With the introduction of new processor chips, such as DEC's Alpha, Motorola's PowerPC, and Intel's Pentium, the race is afoot to offer even greater capabilities and sheer processing power. Computer graphics capabilities and user interface design have also kept pace with ever improving processor hardware. At the same time, hardware has also become more affordable. The combination of these events has provided the opportunity to exploit some of the graphics capabilities and processing power and promote the evolution of the new paradigm of *visual computing*, where concepts can be represented more naturally in a pictorial manner. Visual computing has introduced the concept that the "user interface is becoming a visual representation of the abstract world of the computer." [Singh and Chignell 1992]. Visual computing provides the user with visible models which can be manipulated, thereby reducing the number of unfamiliar abstractions that a user must learn .

2.0 Visual Computing

Visual computing encompasses a wide array of approaches and tools. One useful taxonomy divides visual computing into three main areas: *programming computers, end-user interaction with computers*, and *visualization* [Singh and Chignell 1992] (see Figure 1). This taxonomy is based upon the viewpoint of the user. The area of programming computers (visual programming and program visualization), the first branch of the taxonomy, is the main concern of a program designer. The general end user is most concerned with the user interface, the second branch of the taxonomy. Users who must interpret and process large amounts of data, especially scientists and data analysts, are most interested in the last branch of the taxonomy, namely scientific visualization. The main focus of this paper lies within the first branch of the taxonomy, specifically visual programming which will be discussed at length. The remaining two branches of the taxonomy are both important facets of visual computing both in their present use and future research applications. Their importance warrants a brief introduction of both topics.

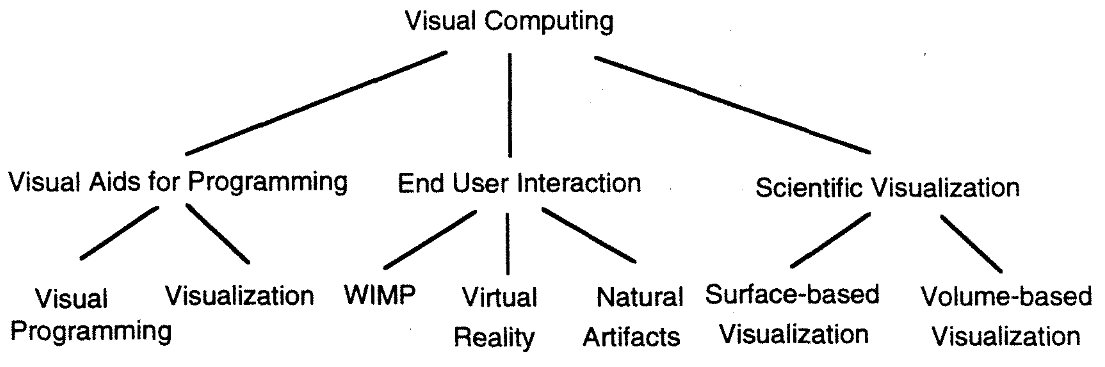


Figure 1. Singh and Chignell's [1992] Classification of visual computing

2.1 Interfacing With Computers

User interfaces have changed dramatically over the last decade. Interfaces have advanced from a string of characters input by the user and output back to the screen by the computer to an interactive manipulation of graphical symbols and the use of new technologies such as head-mounted displays and data gloves. Figure 2 illustrates the taxonomy of end user interaction. [Singh and Chignell 1992]. The first level is based on the technology, both hardware and software, used to implement the interface.

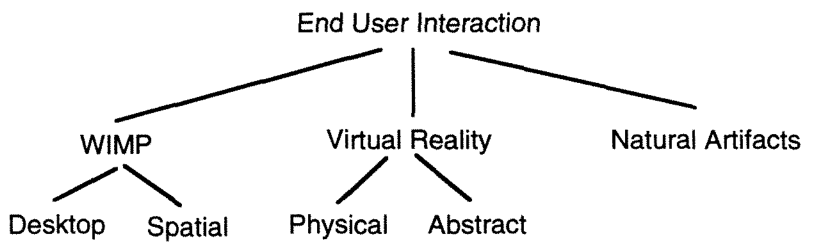


Figure 2. Singh and Chignell's [1992] Taxonomy of end user interaction with computers

The most common technology, by far, is the *WIMP* (Windows, Icons, Menus, and Pointing) interfaces. The taxonomy suggests that WIMP interfaces can be further subdivided into two types of information organization: *desktop* and *spatial*. In the desktop organization, common office objects and operations are recreated on the computer screen, such as filing cabinets, drawers, folders, printers, trash, etc. In the spatial organization, interactions involve moving and manipulating objects within the physical model, such as in Hypertext.

In *virtual reality*, the user is placed within the computer generated environment interfacing with input devices such as data gloves and head-mounted displays to enhance the concept of being inside the environment. Virtual reality has been subdivided into physical reality, where objects are built to behave as their real-world counterparts, and abstract reality, where less tangible information such as energy fields, temperature, and seismic data can be visualized and manipulated.

The final category of end user interaction is *natural artifacts*. This category encompasses communication techniques used in real life such as gestures, handwritten text, and spoken commands.

2.2 Scientific Visualization

The third branch of the taxonomy of visual computing is *scientific visualization*, which enables scientists to map high-volume data into meaningful graphics. "It empowers scientists to investigate the global properties of numerical solutions, examine the dynamics of their data changing over time, interact with the displays to gain further understanding of the data, spot anomalies, or uncover computation errors." [Singh and Chignell 1992]. Scientific visualization can be further subdivided into two main approaches: *surface visualization* and *volume visualization*. Figures 3 and 4 illustrate the use of volume visualization in disease diagnosis [Watson and Watson 1991]. In these examples a multivariate analysis of variance statistical model is used to compare a set of populations on the basis of multiple response variables. Figure 3 shows four population distributions and Figure 4 superimposes the four distributions into one distribution.

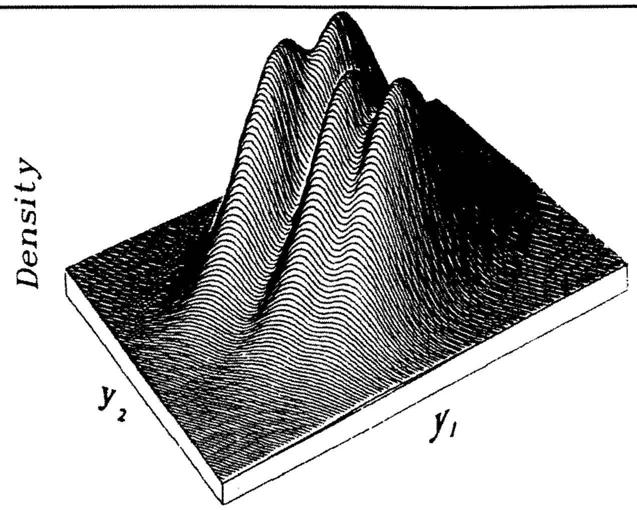


Figure 3. Population Distributions Employed In Computer-Aided Medical Diagnosis [Watson and Watson 1991].

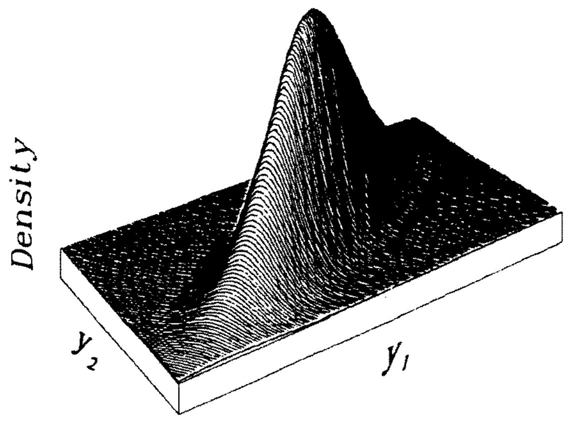


Figure 4. Multiple Population Distributions (of Figure 3) Superimposed Into One Distribution [Watson and Watson 1991].

3.0 Visual Aids for Programming

Returning to Singh and Chignell's [1992] classification of visual computing, let us concentrate on the first branch, *programming computers*. The authors have divided this branch into two key areas: *visual programming* and *visualization* (see Figure 5). The generally accepted definition of visualization is the use of various techniques to aid in the understanding and debugging of computer programs [Baecker and Marcus 1990; Myers 1986; Price *et al* 1993; Singh and Chignell 1992]. Visual programming, on the other hand, "refers to any system that allows the user to specify a program in a two (or more) dimensional fashion. Conventional textual languages are not considered two dimensional since the compiler or interpreter processes it as a long, one-dimensional stream." [Myers 1986]. Baecker and Marcus [1990] offer an especially insightful description of visualization versus visual programming based upon the user's viewpoint.

> "Program visualization focuses on output, on communicating programs, their code, their documentation, their structure, and their behavior to a 'reader' or 'viewer.' Visual programming, on the other hand, focuses on input, on the 'writer' or 'composer' of programs, but usually provides 'feedback' output in the same form as input. Visual programming may appear to subsume program visualization; this is not the case, however, since communicating the structure and process of a program to the reader may need to take advantage of techniques that are not necessarily effective or compatible with those of the writer." [Baecker and Marcus 1990].

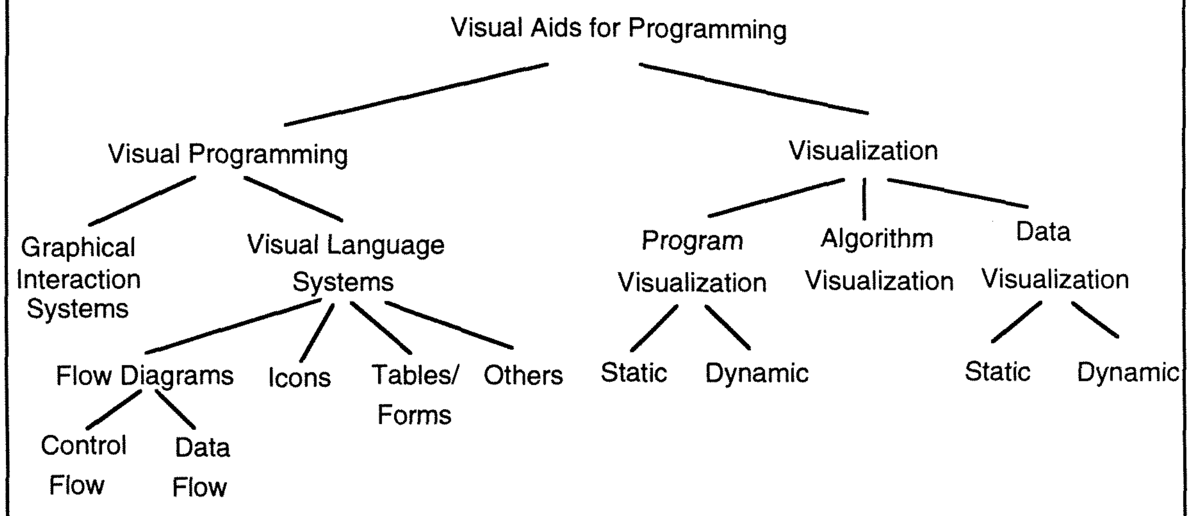


Figure 5. Singh and Chignell's [1992] Taxonomy of visual programming systems

3.1 Visualization

In order to more clearly understand what capabilities a system must process to be classified as a visual programming language, a more detailed discussion of visualization will ensue. As was stated previously, visualization is used to enhance the understanding of computer programs. Singh and Chignell [1992] have further divided visualization into three main branches: *program visualization*, *algorithm visualization*, and *data visualization*.

In program visualization, graphics are used to illustrate some aspect of the program after it is written and can be either *static* or *dynamic program visualization*. Static program visualization techniques include flow charting and *pretty-printing* (insertion of blanks and blank lines, indentation, and comments to enhance the readability of a program). Execution of the program is illustrated either by animation or by highlighting the program code when dynamic program visualization is implemented.

Algorithm visualization systems produce animations of algorithms in order to show the "program fundamental operations that embody both transformations and accesses to data and flow-of-control." [Singh and Chignell 1992].

Data visualization can also be subdivided into either *static* or *dynamic data visualization*. Static data visualization generates static pictorial displays for data structures. This method makes "debugging easier by presenting data structures to programmers in the way that they would draw them by hand on paper." [Myers 1986]. As the name implies, dynamic data visualization graphically displays the state of variables, arrays, lists, trees, and other data structures as the program is executed.

4.0 Visual Programming

In essence, *visualization* provides a means to better understand how a program works after it has been coded. This is in direct contrast with *visual programming*, where a program is actually designed by manipulating graphical representations (icons) or by a combination of icons and textual information.

Singh and Chignell [1992] divide visual programming into two key branches: *graphical interaction systems* and *visual language systems* (see Figure 5). This division is based upon how the graphics are used to build the program. Systems where the user guides or instructs the system in order to create the program are classified as graphical interaction systems. Visual language systems consist of systems in which icons, symbols, charts, or forms are used to specify the program.

4.1 Graphical Interaction Systems

In graphical interaction systems, the sequence of user actions is of vital importance since the system "learns" from the user input. This category is more commonly, and perhaps, more aptly coined *programming by example.*

In the majority of systems, a user is required to specify everything about the program and the system is able to remember the examples for later use. This type of system could be described as *"Do What I Did"* [Myers 1986]. Conversely, some systems attempt to infer the general program structure after the user has provided a number of examples which work through the algorithm. These systems could be characterized by *"Do What I Mean"* [Myers

1986] and are often referred to as automatic programming, which has generally been an area of Artificial Intelligence research.

4.2 Visual Language Systems

The second branch of visual programming is termed *visual language systems*. Within this classification are systems using *icons, symbols, charts*, and *forms* to specify the program. The spatial arrangement of the symbols specifies the program. This differentiates visual languages from graphical interaction systems (programming by example), since, in graphical interaction systems, the user interaction with the system is important, and in visual languages, the arrangement of symbols on the screen is important.

Visual languages are composed of a set of graphical symbols which are constructed into "visual sentences with a given syntax and semantics." [Chang 1987]. Visual sentences must then be spatially parsed to determine the underlying syntactic structure. A semantic analysis must then be performed to determine the meaning of the visual sentences (spatial interpretation). The syntactic and semantic analyses of a visual language differs little from a traditional language approach. Both types of languages must be analyzed to determine syntax and meaning, the significant difference being that visual languages employ graphical symbols rather than textual expressions of traditional languages. In Figure 5, Singh and Chignell [1992] suggest a division of visual languages into three main categories based upon the graphical abstraction used for creating the program: *flow diagrams*, *icons*, and *forms/tables*.

The category, *flow diagrams*, includes visual languages which provide various types charts, graphs, and diagrams to construct programs. Flow diagrams are primarily composed of *control flow* or *data flow diagrams.*

The most common example of control flow diagrams (and probably the earliest visual representation for a program) is the flow chart. Typically, the flow chart was used for documentation purposes, but visual languages employing flow charts create programs automatically. Another type of control flow diagrams used in some visual languages are Nassi-Shneiderman diagrams. Data flow diagrams depict the flow of data from one operation or object to another and the visual language constructs the program from the flow of data.

The second category of visual programming languages, *icons*, consists of visual languages using graphical symbols or icons and their interconnections to form visual sentences. As was noted earlier, spatial parsing and interpretation is used to provide syntactic and semantic analyses, respectively. There is no accepted standard for the definition of an icon. The main criterion for designing an icon is that it clearly represents the abstraction. For example, in LabVIEW, the traditional symbol for an operational amplifier is used to represent the functions add and subtract (see Figure 6). Another use of icons is in the language Proc-BLOX [Chang 1990]. Figure 7 illustrates a Proc-BLOX implementation of a traditional Pascal program, where the Proc-BLOX symbols resemble a three dimensional jigsaw puzzle.

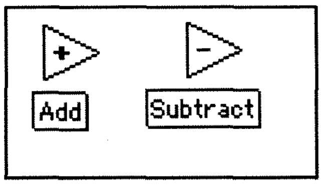


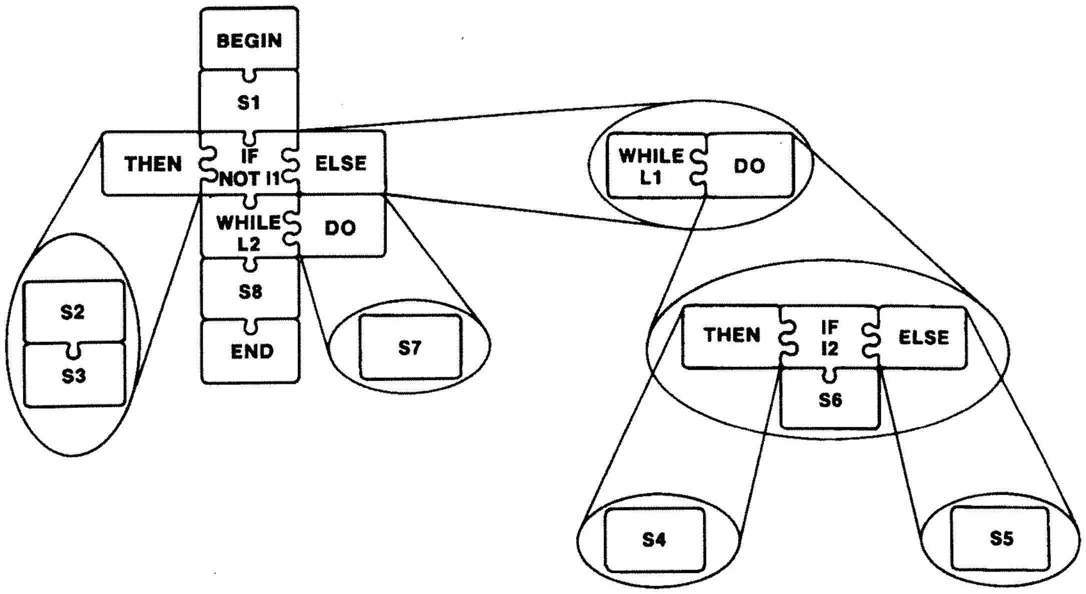
Figure 6. LabVIEW icons for add and subtract functions

The final category of visual languages are those languages which employ *tables* and *forms*. The user constructs the program by using tables or filling in forms. Common uses of this category include developing queries on relational databases through the use of tables and the development of office-information systems using forms.

While the taxonomy designed by Singh and Chignell [1992] offers a strong basis for distinguishing aspects of visual computing, it could be enhanced by incorporating Chang's [1987] more rigorous approach.

Chang [1987] classifies four types of visual languages: *languages supporting visual interaction, visual programming languages, visual information processing languages*, and *iconic visual information processing languages.* This classification is based upon the objects which the language processes, the transformation of the object, and how the language constructs are represented. In Figure 8, an object icon is represented as a two-part entity, written as (Xm, Xi), where Xm represents the meaning or logical part and Xi represents the visual image. Languages supporting visual interaction and visual programming languages deal with objects that have logical meaning but no visible image: (Xm, e), where e denotes a null object. In order for the object to be visualized, we must transform (Xm, e) into (Xm, X'i). Similarly, languages dealing with inherently visual objects with no logical meaning must transform (e, Xi) into (X'm, Xi). Further classification is based upon whether the language constructs are visual or linear.

Languages which support visual interaction process objects that do not have an inherent visual representation, such as data types (arrays, stacks, queues, etc.) and application data types (forms, documents, databases, etc.). These languages use iconic representations such as entity-relationship diagrams, action diagrams, and Nassi-Shneiderman diagrams, but the program statements are still written in a conventional programming language.



```
BEGIN
    S1;
    IF NOT I1 THEN
        BEGIN
            S2;
            S3;
        END
    ELSE
        WHILE L1 DO
            BEGIN
                IF I2 THEN S4 ELSE S5;    S6;
            END;
    WHILE L2 DO S7;
    S8;
END;
```

Figure 7. Proc-BLOX Representation and Pascal Program Equivalent [Chang 1990]

Type of visual language	**Objects to be dealt with**	**Transformation of objects**	**Languages' visibility**
Languages that support visual interaction	logical objects with visual representation	(Xm, e) --> (Xm, X'i)	linearly represented constructs
Visual programming languages	logical objects with visual representation	(Xm, e) --> (Xm, X'i)	visually represented constructs
Visual information processing languages	visual objects with imposed logical representation	(e, Xi) --> (X'm, Xi)	linearly represented constructs
Iconic visual information processing languages	visual objects with imposed logical representation	(e, Xi) --> (X'm, Xi)	visually represented constructs

Figure 8. Chang's [1987] Four Types of Visual Languages

As with languages supporting visual interaction, visual programming languages also process objects which do not have an inherent visual representation but are transformed from (Xm, e) into (Xm, X'i) to be represented visually. Not only are the objects represented visually, but the rules for combining these objects are also represented visually. Some applications of visual programming languages include computer graphics, user interface design, database interface design, form management, and computer-aided design.

A visual information processing language processes objects with an inherent visual representation and are transformed from (e, Xi) into (X'm, Xi) to impose a logical interpretation. Typically, these languages are implemented in a linear language with an enhanced user interface to accommodate the graphical images. Applications of visual information processing languages include image processing, computer vision, robotics, image database management, office automation, and image communications.

Iconic visual information processing languages are differentiated from visual information processing languages by the fact that iconic languages also represent language constructs visually and not linearly.

5.0 Visual Language Evaluation

A thorough evaluation of a visual language is a daunting task. Since the paradigm has just been recently introduced, or more aptly, recently discovered by the general populace, scant research has been published on the subject of visual language evaluation. One exception is Shu's three-dimensional framework to characterize and compare visual languages as illustrated in Figure 9 [Shu 1988]. Shu's three criteria for comparison are *Visual Extent, Scope,* and *Language Level. Visual Extent* is a measure of the language's use of graphical objects for programming constructs. *Scope* is an indicator of whether a visual language is applicable only in a very limited area or useful in a variety of applications. Finally, *Language Level* displays whether the visual language qualifies as a high or low level language. Presently, there is no standard classification scheme for classifying visual programming language research papers, however, Burnett and Baker [Burnett and Baker 1993] have recommended such a classification (see Figure 10). Although Burnett and Baker 's proposed classification of visual programming languages is intended for research paper classification, it presents several pertinent attributes of a visual programming language. A programming language can be determined as visual based upon the possession of these attributes. Furthermore, these attributes can then be extended and used as a basis for comparison of visual programming languages. Within the same proposal, Burnett and Baker present an overall Visual Computing hierarchy (see Figure 11), where their paper classification proposal focuses on the branch labeled Visual Programming Languages.

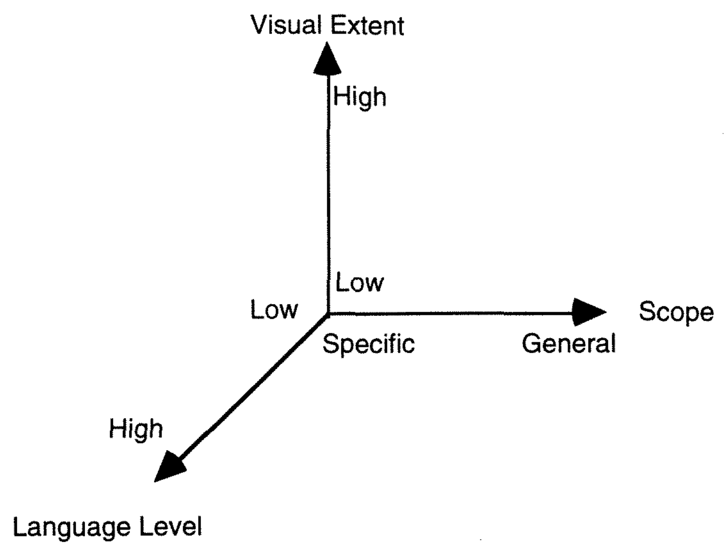


Figure 9. A three-dimensional framework to characterize and compare visual languages [Shu 1988]

A critical set of visual programming language evaluation criteria has been developed based upon concepts from Shu's characterization, Burnett and Baker's proposed visual language research paper classification, and extensive evaluation of programming languages. This set of criteria and accompanying explanation can be found in Figure 12. The criteria were developed specifically for visual programming languages with the intent that visual languages can be evaluated and reevaluated with the evolution of the paradigm. In general, the evaluation criteria range of measures will include:

- **none** - attribute not implemented
- **weak**- language provides functionality to accomplish attribute but is poorly implemented either because the language imposes strict adherence to detailed requirements or the task cannot be accomplished directly.
- **fair** - attribute is implemented but limited in its ability
- **strong** - attribute fully implemented

Other attributes are more easily characterized by the terms specific and general, both of which are self-explanatory. A list of possible values are provided in the accompanying definition for attributes which can be assigned certain values.

Many languages touted as 'visual programming' languages, however, do not possess the attributes to be classified as a visual programming language. For example, Visual Basic and Visual C++ would be placed within the End User Interaction branch of Singh and Chignell's classification of Visual Computing. These languages do not fall within Burnett and Baker's Classification of Visual Programming Language Research, rather these languages would be placed within the Visual Environments for Textual Languages in their overall view of Visual Computing. Although these languages are often termed as 'visual programming' languages, they are more aptly classified as textual languages with strong graphical user interface capabilities. In these Windows applications, interactive screens can be easily generated. This is accomplished through choosing an icon (either a standard icon or a user-generated icon), placing it in the desired screen position, setting the parameters, then programming **textually** what events will occur when that icon is chosen during execution. A limited amount of graphical animation can also be implemented within these systems, such as the sequencing of a set of graphical images. These languages do not provide a facility for algorithm animation. Although these languages greatly simplify the task of building a Windows application, they are not considered to be visual languages since coding is accomplished textually.

VPL: Visual Programming Language

VPL - I. Environments and Tools for VPLs

VPL- II. Language Classifications
- A. Paradigms
  1. Concurrent Languages
  2. Constraint-based languages
  3. Data-flow languages
  4. Form-based and spreadsheet-based languages
  5. Functional languages
  6. Imperative languages
  7. Logic languages
  8. Multi-paradigm languages
  9. Object oriented languages
  10. Programming-by-demonstration languages

     . . .
- B. Visual representations
  1. Diagrammatic languages
  2. Form-based and spreadsheet-based languages
  3. Iconic languages
  4. Languages based on static pictorial sequences

VPL- III. Language Features
- A. Abstraction
  1. Data Abstraction
  2. Procedural abstraction
- B. Control flow
- C. Data Types and structures
- D. Documentation
- E. Event handling
- F. Exception handling

VPL-IV. Language Implementation Issues
- A. Computational models
- B. Efficiency
- C. Parsing
- D. Translators (interpreters and compilers)

VPL-V. Special-Purpose Languages
- A. Database languages
- B. User-interface generation languages
- C. Image-processing languages
- D. VPLs for scientific visualization

  . . .

VPL-VI. Theory of VPLs
- A. Formal definition of VPLs
- B. Icon theory
- C. Language design issues
  1. Cognitive and user-interface design issues
  2. Liveness
  3. Scope
  4. Effective used of screen real estate
  5. Type checking and type theory
  6. Visual representation issues

     ...

Figure 10. Burnett and Baker's [1993] Proposed Visual Programming Research Paper Classification.

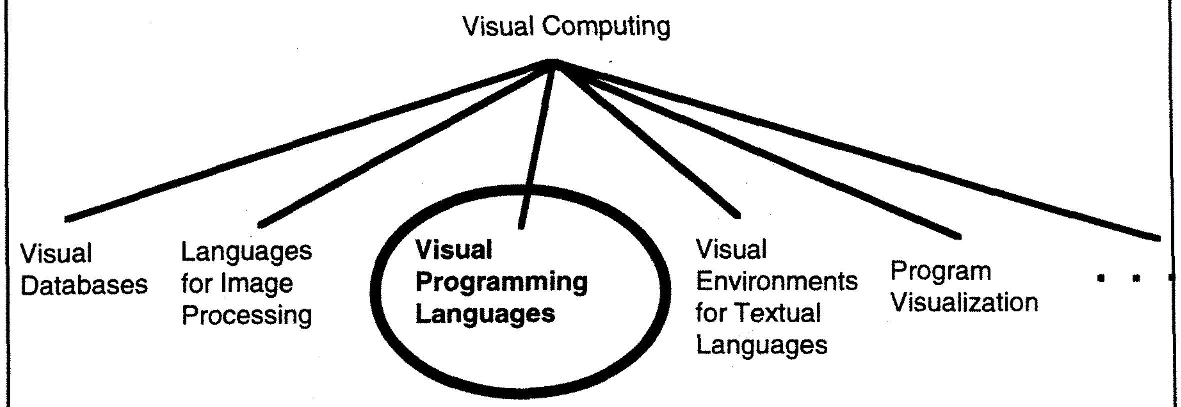

Visual Computing
Visual Databases
Languages for Image Processing
Visual Programming Languages
Visual Environments for Textual Languages
Program Visualization
. . .

Figure 11. Burnett and Baker's [1993] Visual Computing Hierarchy

There are a limited number of commercially available visual programming languages. Two visual languages which are available commercially are National Instrument's LabVIEW and The Gunakara Sun Systems' ProGraph. The features, strengths, and weaknesses of these two application software packages will be explored.

Visual Language Attribute	Definition
• **Scope**	The applicability of the software package to various applications, ranging from **specific** to **general.**
• **Intended Audience**	What group of users would most benefit from the incorporation of the visual language, ranging from **specific** to **general**.
• **Paradigm**	What programming paradigm is implemented. Possible values include: **dataflow**, **object-oriented**, **object-oriented dataflow**, **structured**.
• **Ease of Use**	How quickly the software package can be adopted to develop an application, ranging from **weak** to **strong**.
• **Visual Representation**	How the graphical language is visually presented. Possible values include: **complex iconic structure**, **simple iconic structure**, **tables**, **forms**, **flow diagrams**.
• **Compiler**	What type of compiler is implemented, ranging from **graphical** (direct compilation) to **interpreted** (compiled to intermediate code).
• **Reusability**	How easily can code be reused, ranging from **weak** to **strong**.
• **Data Types and Structures**	How extensive are the system defined and user-defined types, ranging from **weak** to **strong**.
• **Effective Use of Screen Area**	How well can screen space be conserved, ranging from **weak** to **strong**.
• **Hardware**	For what hardware platform(s) is the software available. Possible values include: **Macintosh**, **IBM**, **Sun**.
• **Operating System**	Under what operating systems does the software run. Possible values include: **Macintosh OS**, **Windows**, **Sun OS**.
• **Animation (runtime visualization)**	How extensive is the runtime visualization of algorithm execution, ranging from **none** to **strong**.
• **Effective Use of Colors**	How well are colors used to depict different graphical components, ranging from **none** to **strong**.
• **Clarity of graphical symbols**	How easily recognizable graphical symbols are, ranging from **weak** to **strong**.
• **Interactive Capabilities**	Measure of ability to modify program settings during execution, ranging from **none** to **strong**.
• **Extensibility**	How easily the program can be extended to include upgrades or modifications, ranging from **weak** to **strong**.
• **Interface Capabilities with Other Languages**	How extensive are the facilities provided to link with other languages or packages, ranging from **none** to **strong**.
• **Analysis Capabilities**	How extensive are data analysis capabilities are included in the programming language for general use, ranging from **none** to **strong**.

Figure 12. Visual Language Evaluation Criteria and Definitions.

6.0 Comparison of Two Visual Languages: LabVIEW and ProGraph

National Instrument's LabVIEW and The Gunakara Sun Systems' ProGraph are both iconic visual programming languages. Their position in Singh and Chignell's taxonomy for Visual Languages would be in the Icons category as demonstrated in Figure 13. LabVIEW and ProGraph have been classified as iconic languages *employing* the dataflow paradigm rather than as merely dataflow diagram systems. This distinction is made since both languages provide an extensive library of icons. These systems were compared using Macintosh II hardware and the Macintosh operating system.

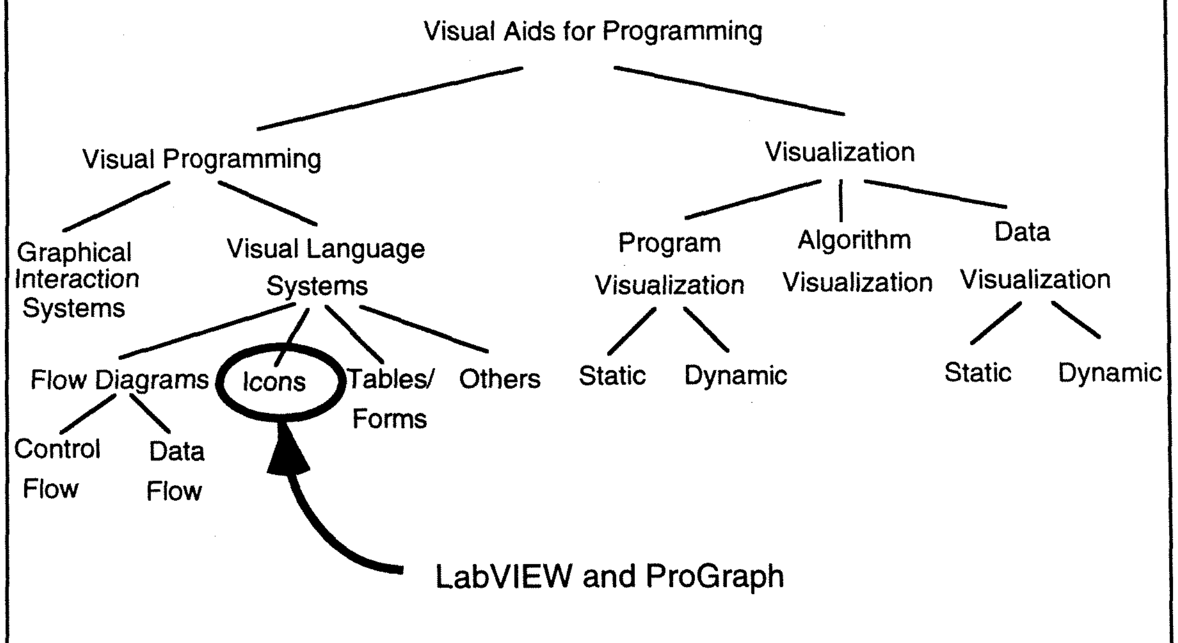


Figure 13. LabVIEW and ProGraph Position in Singh and Chignell's [1992] Taxonomy.

LabVIEW and ProGraph are compared using Shu's three-dimensional characterization scheme as shown in Figures 14 and 15. LabVIEW ranks extremely high in the area of Visual Content and fairly high in the Language Level. LabVIEW is software package for instrumentation, thereby ranking specific in the Scope area. In contrast, ProGraph's Visual Content is not nearly as extensive and ranked low in that area. ProGraph's strengths are that it is a high-level object-oriented language useful in general applications and ranks extremely high in Language Level and general in Scope.

In Figure 16, LabVIEW and ProGraph are compared using the previously defined evaluation criteria. Following this table of comparisons, each language will be more thoroughly introduced. First, a general overview will be presented for LabVIEW, followed by a more detailed description of each evaluation criterion. An example LabVIEW application will then be described. This same format will be maintained for ProGraph.

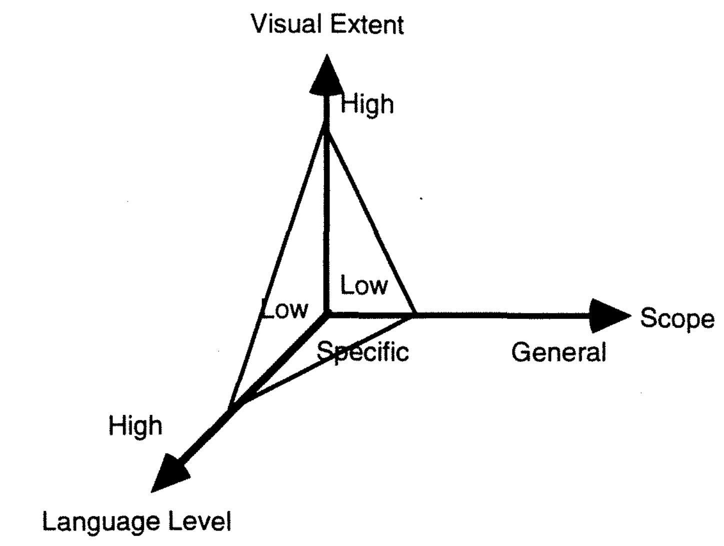


Figure 14. LabVIEW 2 Three-Dimensional Evaluation.

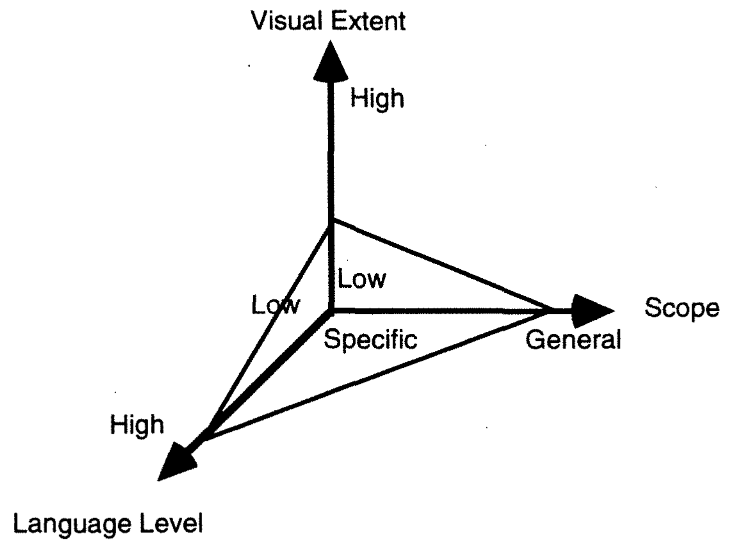


Figure 15. ProGraph Three-Dimensional Evaluation.

Attribute	LabVIEW	ProGraph
• **Scope**	**specific -** instrumentation applications	**general -** general object-oriented programming applications
• **Intended Audience**	**specific -** engineering personnel	**general -** general programming audience
• **Paradigm**	**dataflow**	**object-oriented dataflow**
• **Ease of Use**	**strong -** initial use very intuitive - more involved implementations not very intuitive	**fair -** basic object-oriented concepts presented well - building applications not as intuitive
• **Visual Representation**	**complex iconic structure**	**simple iconic structures-** with textual annotations
• **Compiler**	**graphical**	**graphical**
• **Reusability**	**strong -** developed Virtual Instruments can be reduced to an icon to be used in other applications	**strong -** since there is inheritance, methods and attributes from ancestors can be easily reused an modified
• **Data Types and Structures**	**weak -** no capability for user defined classes or structures	**strong** extensive ability to define structures with user defined classes and inheritance
• **Effective Use of Screen Area**	**fair -** any portion of a VI can be iconized to conserve screen space but some standard icons cannot be resized	**strong -** any portion of a method can be reduced to a local method to conserve screen space
• **Hardware**	**Macintosh, IBM, Sun**	**Macintosh**
• **Operating System**	**Macintosh OS, Windows, Sun OS**	**Macintosh OS**
• **Animation (runtime visualization)**	**strong -** data can be tracked as it flows through diagram	**strong -** data can be tracked as it flows though methods as well as the system stack can be monitored
• **Effective Use of Colors**	**strong -** data types are color-coded as in a physical electrical schematic	**none**
• **Clarity of graphical symbols**	**strong -** most of the simple icons are highly intuitive, but more complicated icons require extensive probing	**fair -** most of the symbols are not intuitive but are textually annotated
• **Interactive Capabilities**	**fair -** front panel controls can be modified during execution but diagram cannot be rewired during execution	**strong -** if an error is encountered during execution, a correction can be made and execution will resume from the stop point
• **Extensibility**	**fair -** can require extensive rewiring of blocks	**strong -** object-oriented paradigm enables new classes and methods to be added easily
• **Interface Capabilities with Other Languages**	**weak -** data can be saved in standard format for use in other software and compiled C code can be imported, with great difficulty	**strong -** compiled C code can be incorporated into the application
• **Analysis Capabilities**	**strong -** digital signal processing, numerical, and statistical analysis options	**weak -** software not designed for analysis capabilities

Figure 16. Comparison of Visual Language Attributes for LabVIEW and ProGraph.

6.1 LabVIEW

6.1.1 Overview

LabVIEW provides an extensive choice of iconic programming structures, from simple mathematical functions to complicated Digital Signal Processing. All icons are chosen through a typical point and click Macintosh menuing system The iconic functions are classified logically, therefore determining in which menu an iconic function is located is a trivial task. However, not all iconic representations are easily interpreted. Figures 17, 18 and 19 display the menu choices of arithmetic, trigonometric and logarithmic, and comparison icons, with most of the icons in each menu being easily recognizable. If an icon in these categories is not immediately apparent, the accompanying short description displayed when an icon is highlighted is typically sufficient to determine its functionality.

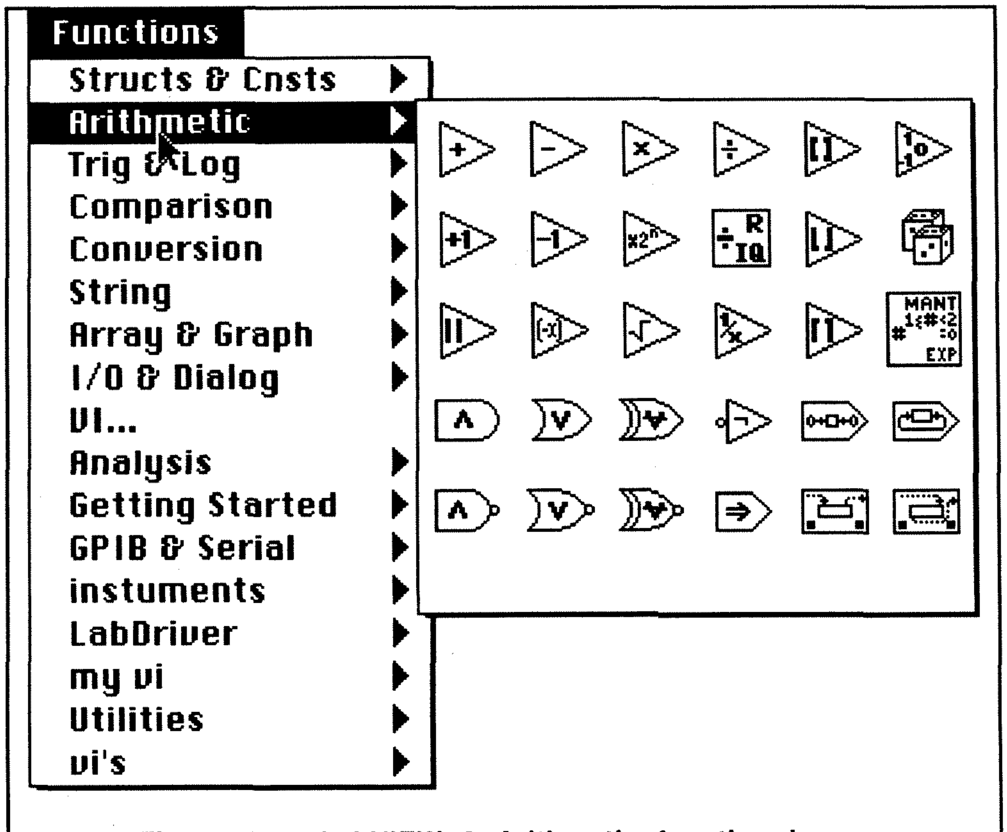


Figure 17. LabVIEW 2 Arithmetic function icons.

Not all icons' functions, however, are immediately recognizable. For example, Figure 20 is a snapshot of the structures and constants menu. Many of the constants are self-explanatory, but the programming structures themselves are not readily apparent. A brief description of the function of the programming structures is presented in Figure 21. Once the icon is defined, the general shape of the icon does then seem to accurately reflect the function. As the function of the icon becomes more complex, the meaning of the icon becomes more obscure, especially in the Analysis menu selection.

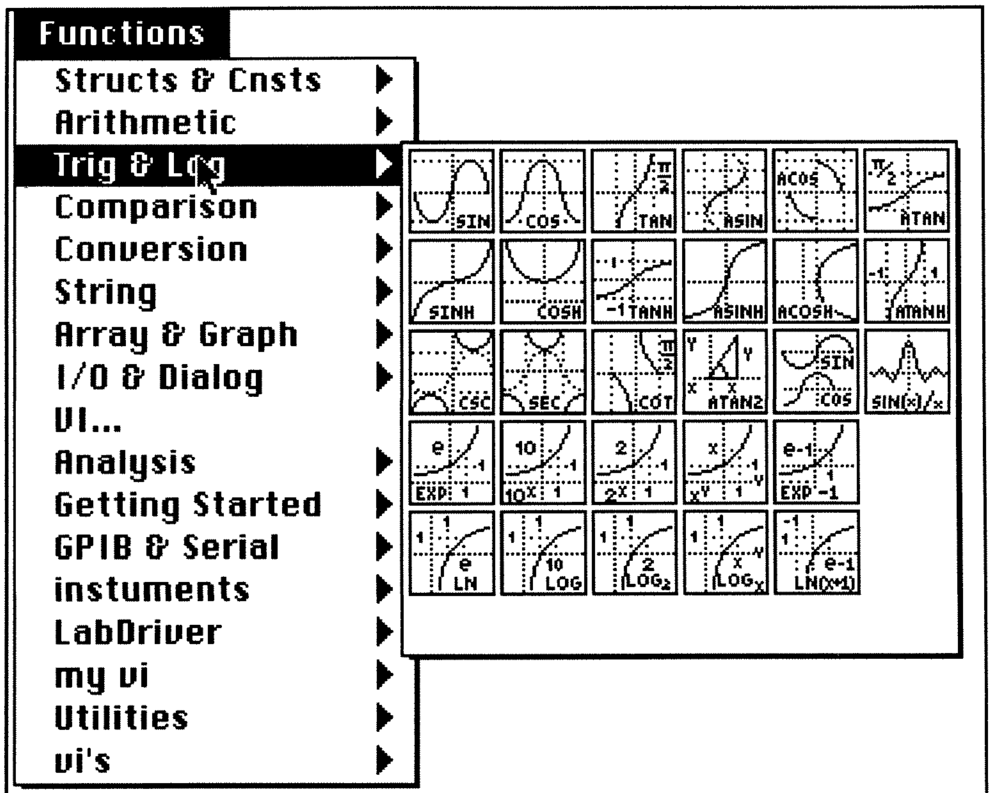


Figure 18. LabVIEW 2 Trigonometric and Logarithmic function icons.

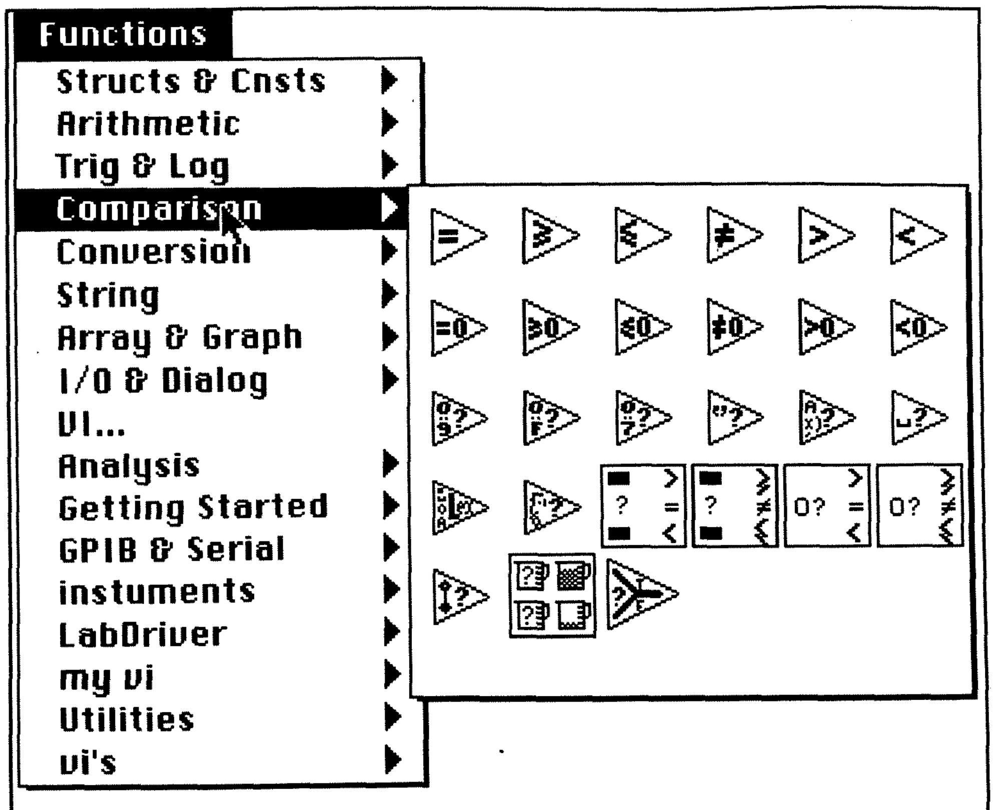


Figure 19. LabVIEW 2 Comparison function icons.

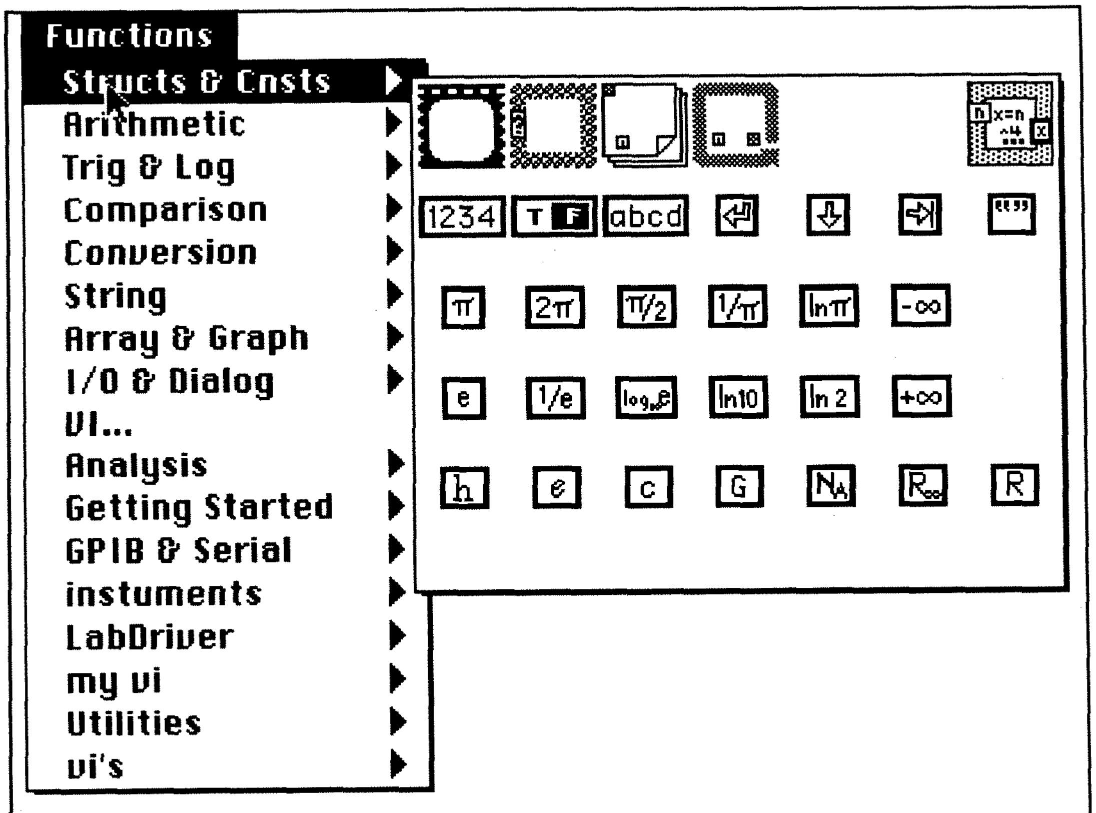


Figure 20. LabVIEW 2 Structures and Constants Icons.

The sequence structure (see Figure 21), forces the execution of diagrams in a specific order. Since LabVIEW is a dataflow language, a block will execute as soon as all of its inputs are available. The sequence structure provides the method to override this feature. The for loop, while loop, and case structures function the same as in a traditional language. In the while loop icon, the structure wired to the circular arrow located within the while loop icon is the condition statement. The element wired to the question mark in the case structure will determine which case diagram to execute.

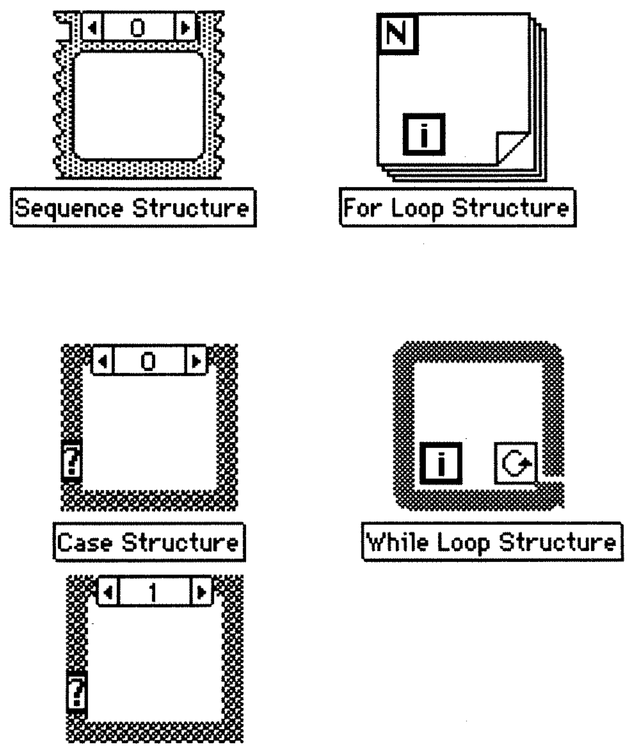


Figure 21. LabVIEW Sample Structures.

LabVIEW has extensive data analysis capabilities. To demonstrate LabVIEW's power, the Digital Signal Processing (DSP), Numerical, and Statistical menu choices are presented in Figures 22 through 25. These analysis capabilities cover essentially all test system design requirements. They can be incorporated into a Virtual Instrument as real-time analysis without necessitating porting data to another data analysis application. A Virtual Instrument designed to highlight LabVIEW's analysis functions will be presented in detail in a later section.

The Digital Signal Processing (DSP) selection of the Analysis menu offers a wide variety of digital signal processing options (see Figure 22). Waveforms, such as impulse, pulse, ramp, triangle, sine, and square waves, can be generated and placed into an array or displayed on a graph. Noise signals can be generated and added to another signal. The Fast Fourier Transform, Hartley Transform, and Hilbert Transform can be used to convert a time domain waveform into its corresponding frequency domain representation. Other DSP functions include determining the power spectrum of a waveform, integrating a waveform, and determining the first derivative of a waveform at a specific point.

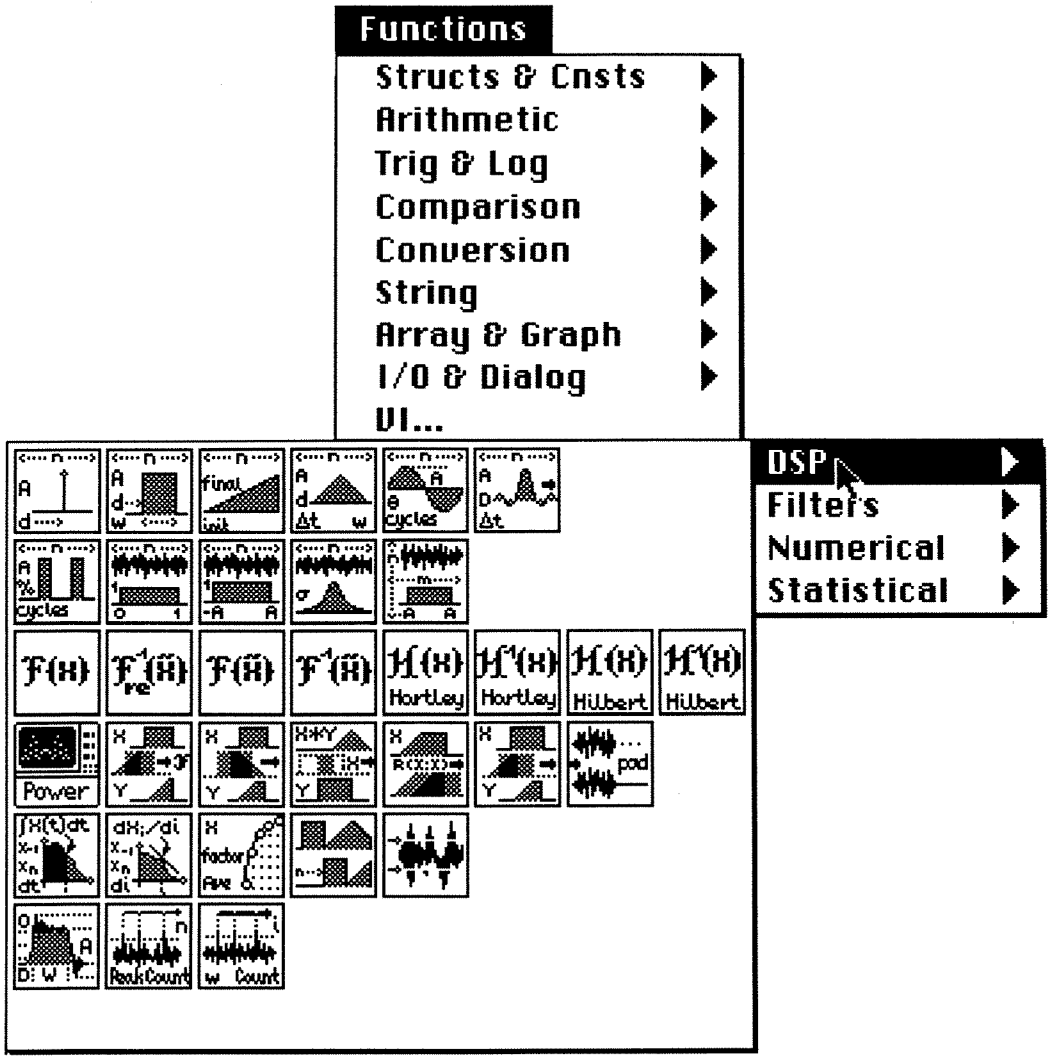


Figure 22. LabVIEW 2 Digital Signal Processing Icons.

LabVIEW provides many common signal filtering functions. These filters include removing of spikes from waveforms, removing high frequencies from a waveform (low pass filters), removing low frequencies from a waveform (high pass filters), and removing both high and low frequencies (band pass filters). Figure 23 lists the icons of all available filters.

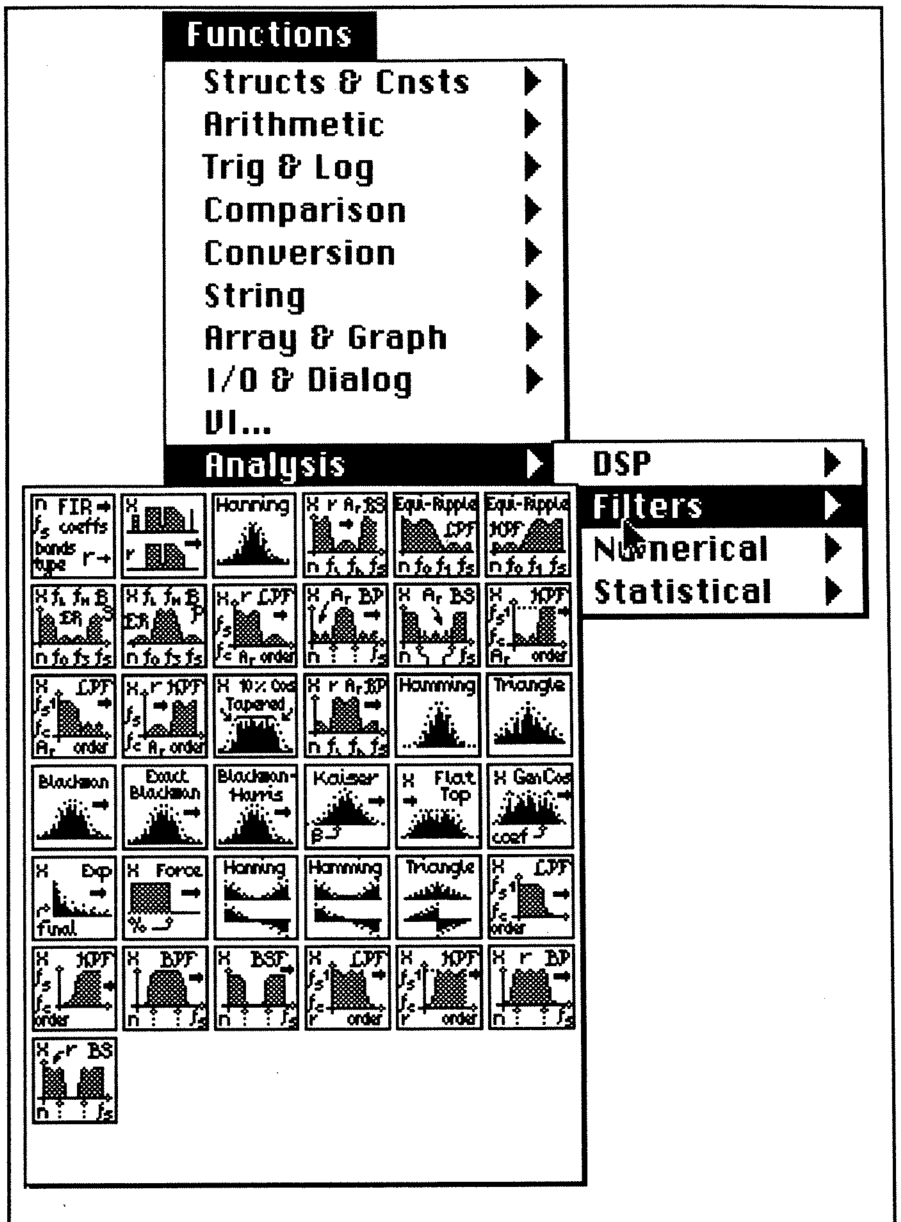


Figure 23. LabVIEW 2 Filter Icons.

Figure 24 lists the numerical functions from the analysis menu selection. The numerical functions include returning the x- and y-components of a vector, separating the real and imaginary components of an array, determining the base-10 log and natural log of each element in an array, as well as taking the base-10 log of each array element and multiplying by 10 or 20.

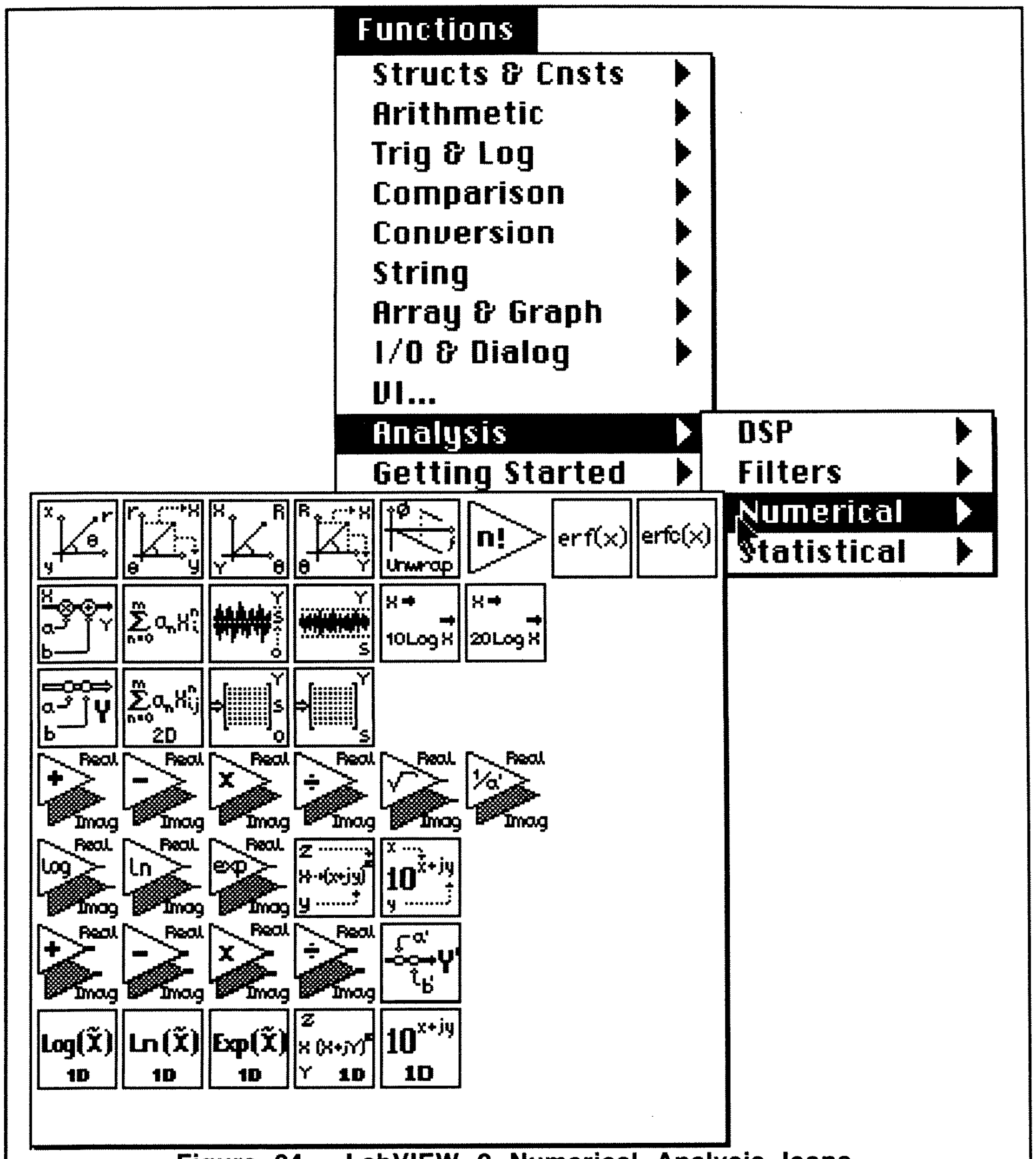


Figure 24. LabVIEW 2 Numerical Analysis Icons.

The statistical functions of the analysis menu selection (see Figure 25) offers many typical capabilities. Icons can provide such functions as determining array mean, standard deviation, RMS (root mean square) of values, fitting waveforms both linearly and polynomial, dot and cross products of matrices, and solving linear equations.

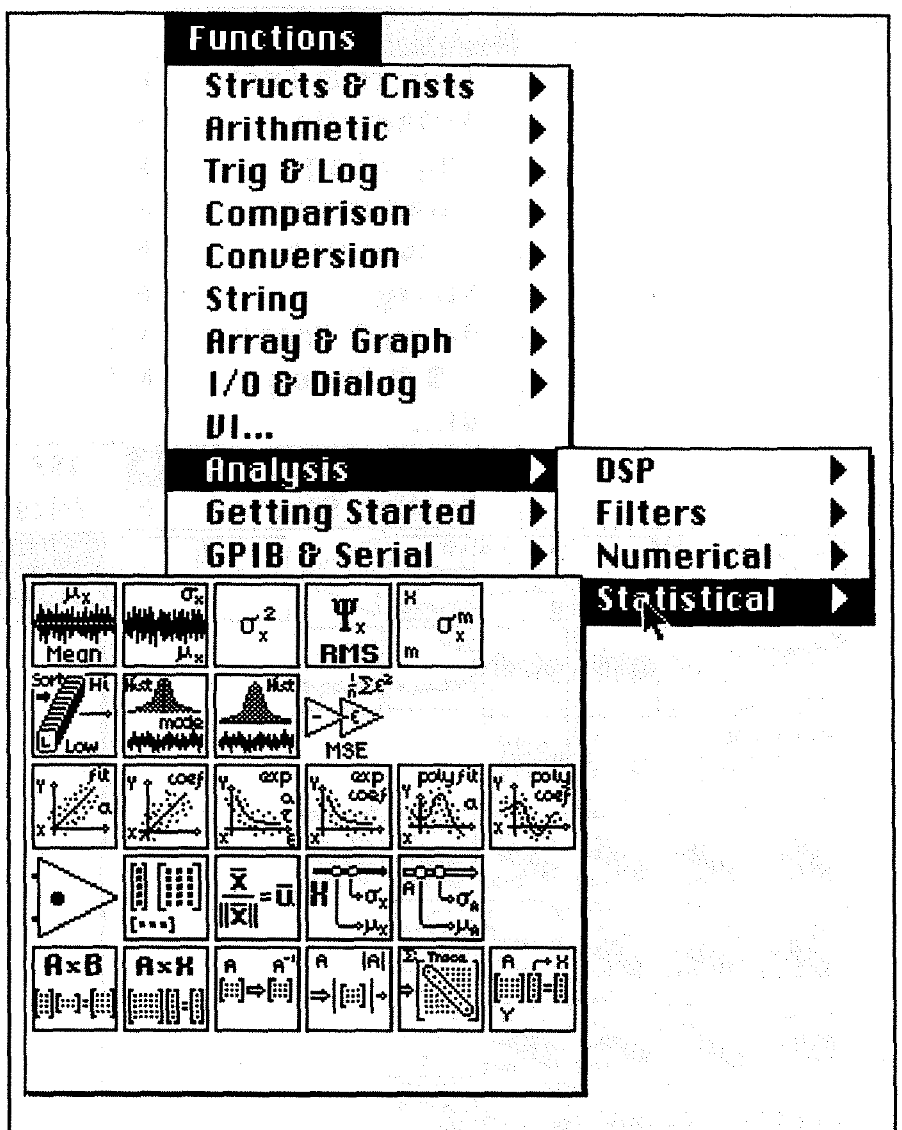


Figure 25. LabVIEW 2 Statistical Analysis Icons.

6.1.2 Scope

LabVIEW has been designed as a graphical programming system for data acquisition and control, data analysis, and data presentation. Its scope is specifically designed for instrumentation of test and measurement applications. Software modules are assembled graphically and are termed *Virtual Instruments* (VIs). The Virtual Instrument is built to acquire data from plug-in data acquisition boards, IEEE-488 and RS-232 programmable instruments, perform (potentially extensive) data analysis, and present the results through graphical user interfaces.

Certainly, some general purpose programs can be constructed using LabVIEW. The available data structures are fairly simple, therefore limiting its scope. It is an especially useful tool for the introduction of the visual programming paradigm. These uses, however, obscure its true strength - instrumentation design. After mastering the meaning of the graphical symbols, the difficulty of interfacing instruments to produce an automated system is significantly reduced. An engineer no longer needs to remember cryptic IEEE-488 (GPIB) codes to control an instrument. The library of available VIs for commercial instruments is quite extensive and includes many of the most common instruments.

A VI is composed of a *front panel*, a *block diagram*, and an *icon/connector*. The front panel is the user interface, the block diagram is the VI source code, and the icon/connector is the calling interface. A block diagram contains input/output, computational, and subVI components, which are represented by icons and interconnected by lines directing the flow of data. Input/output components communicate directly with external physical instruments. Computational components perform arithmetic and other operations. SubVI components call other VIs, passing data through their icon/connectors.

6.1.3 Intended Audience

LabVIEW was designed with instrumentation engineers and technicians as the intended audience. The terminology and graphical representations used are consistent with the engineer's vocabulary. Many of the functions, such as adding, subtracting, multiplying, dividing, and comparison operators, are drawn as an operational amplifier - a symbol familiar to the engineer. Logic functions are represented by the traditional symbols of logic circuit design. Program construction is accomplished by connecting icons using a wiring tool, so programs are designed by wiring elements in a very similar manner to a circuit schematic diagram.

The front panel icons are reproductions of physical components engineers would employ in the construction of an automated testing or process control systems. A variety of control icons and indicator icons are displayed in Figures 26 and 27, respectively. Control icons allow the user to change values during execution, whereas indicator icons only display relevant information without opportunity for operator input.

6.1.4 Paradigm

LabVIEW is based upon *dataflow programming*, where each node begins execution only when data is available at all of its inputs. This paradigm allows for creation of diagrams with independent or parallel dataflow paths and simultaneous operation.

6.1.4.1 Dataflow Programming

Dataflow programming is based entirely on the concept of data flowing from one function to another. In dataflow programming, data flow through the program activating each instruction as soon as all the required input data have arrived. In contrast, the traditional von Neumann architecture programming language is based upon manipulating the state of a global memory using the sequential execution of a set of language commands. Dataflow programming is not limited to sequential execution of instructions. "The dataflow paradigm allows for more than one instruction to be executed simultaneously. The concurrency in dataflow execution depends purely on the availability of data at instruction-execution time, the proportion of concurrency specified in the application to begin with, and how sufficient the computing resources are for handling concurrent executions. Because of this, dataflow programs are said to allow for fine-grain concurrency at the instruction level of a program." [The Gunakara Sun Systems 1992A].

Although LabVIEW is a dataflow programming language, instructions can be forced to execute in a specific manner if so desired. The structure to order execution is referred to as a *sequence structure*.

Some literature has touted LabVIEW as an object-oriented programming language. Although LabVIEW focuses the designer's views on physical entities such as instruments and their measurements (objects), there is no mechanism for inheritance or polymorphism. By most definitions of object-oriented programming languages, LabVIEW would not be classified as object-oriented, but perhaps as object-based.

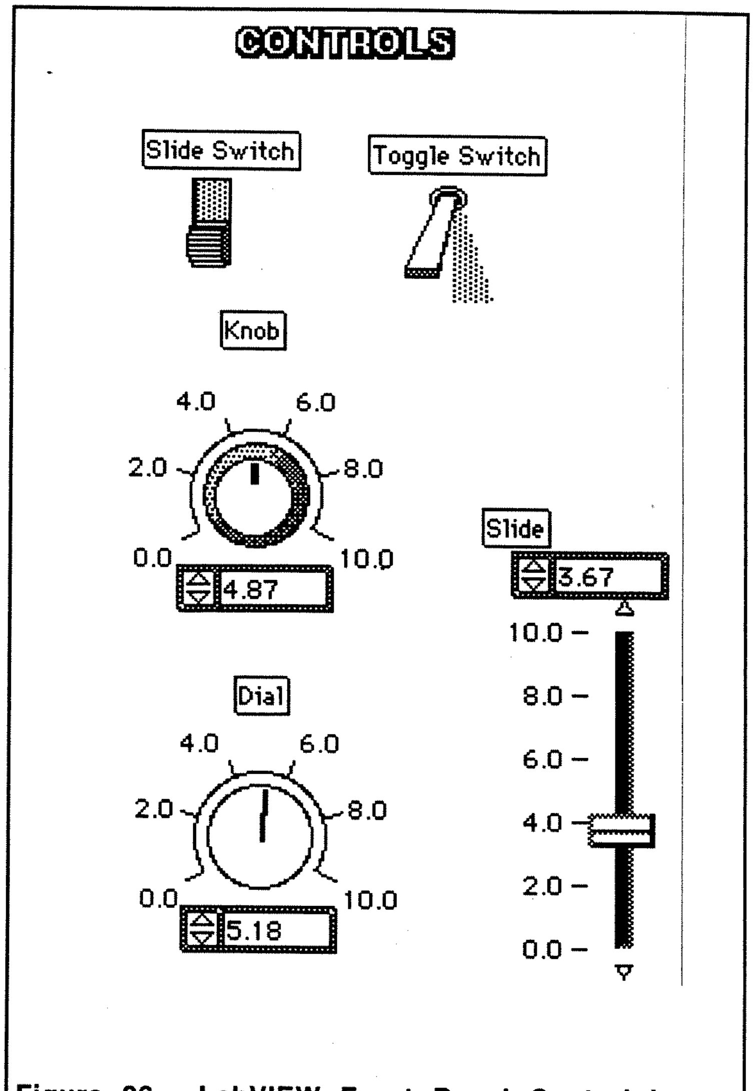


Figure 26. LabVIEW Front Panel Control Icons.

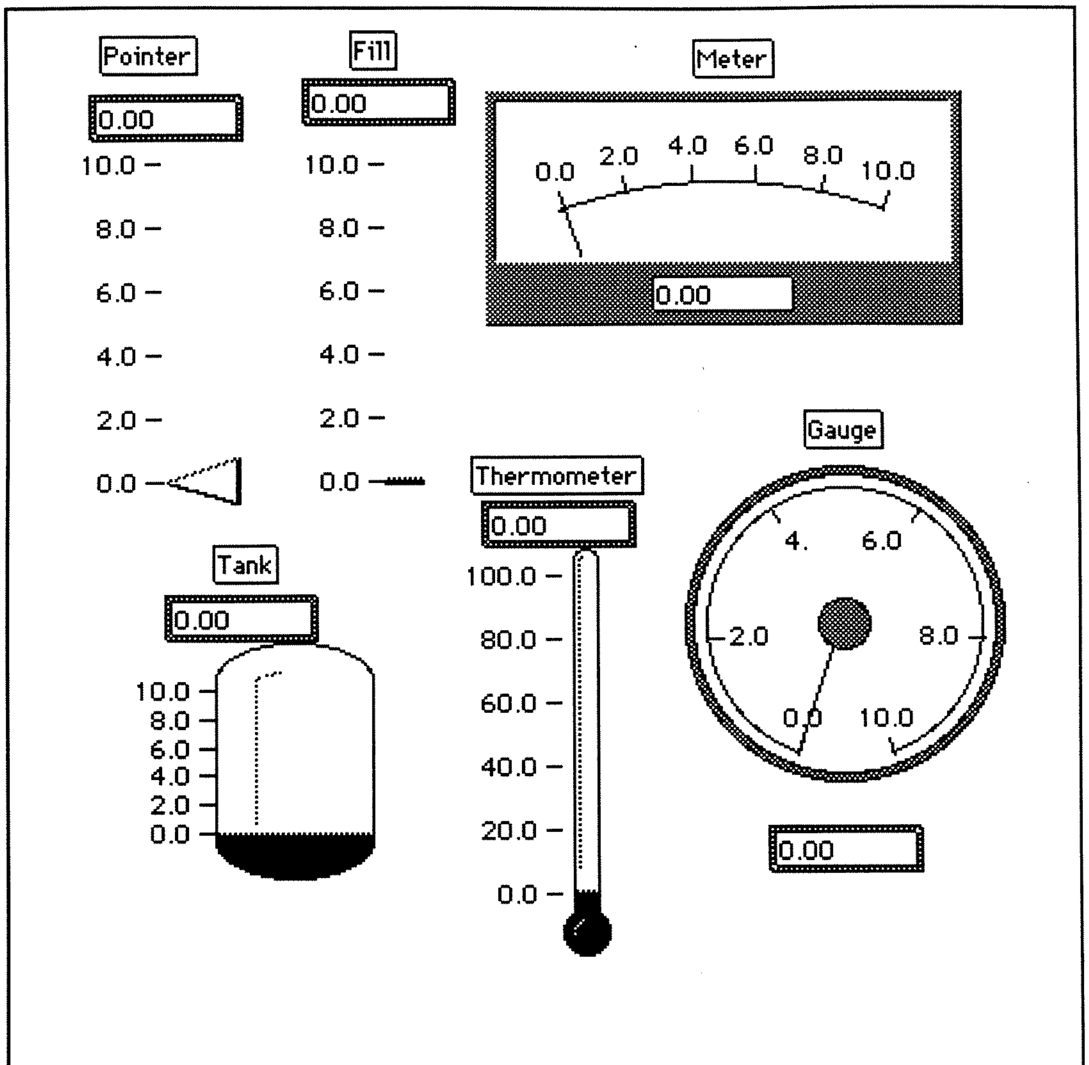


Figure 27. LabVIEW Front Panel Indicator Icons.

6.1.5 Ease of Use

Simple Virtual Instrument applications can be developed in LabVIEW rather quickly. The front panel icons are representations of physical entities and are easily recognizable. Many of the block diagram icons are very intuitive and the function of less recognizable icons can often be determined by the accompanying name in the pull down menu. The help facility also provides information on the required inputs and provided outputs of block diagram icons. With the help of the *Getting Started Manual* [National Instruments 1990A], more complicated applications can be built.

6.1.6 Visual Representation

LabVIEW is an iconic visual programming language. Many of the icons provide highly complex functions. This built-in complexity eases the task of performing detailed analytical computations. In order to use an icon, the proper data must be wired to inputs and wired from the outputs.

6.1.7 Compiler

LabVIEW VI source code **is** the block diagram. This block diagram is compiled directly into machine code, thus the compiler is termed as graphical.

6.1.8 Reusability

Once a Virtual Instrument has been constructed, it can be 'iconized' and used as a subVI in any other program. This provides a high degree of reusability. SubVIs can be used in an hierarchical fashion. In other words, VI1 can call subVI2, which in turn calls subVIs 3 and 4, which can then call other subVIs. This modular, hierarchical design promotes reusability throughout applications.

6.1.9 Data Structures and Types

Data structures and types are very limited in LabVIEW. Using the same data structure in different parts of a program is difficult. Also, there is no facility for user-defined types or structures. For example, there is no inherent method to link components of arrays of different types. Multiple dimension arrays can be defined, but all columns must be of the same data type. In most engineering applications, the crucial data is invariably numerical, thus LabVIEW has concentrated its data typing specifically for engineering applications.

6.1.10 Effective Use of Screen Area

Any Virtual Instrument can be iconized and used as a subVI, which can be inspected by double-clicking on that icon. Even with the ability to use subVIs, it can be difficult to develop a complicated application in a limited space. The diagram and front panel can extend beyond the size of the screen and scroll bars used to view the additional area. Unfortunately, certain standard icons cannot be resized to conserve space, such as the analysis icons illustrated in Figures 22 through 25. Also, it is not a simple task to choose a portion of a VI to be made into a subVI.

6.1.11 Hardware

LabVIEW is available to run on the following hardware platforms: Macintosh, IBM, and Sun.

6.1.12 Operating Systems

LabVIEW is available to run under the Macintosh operating system, Windows, and the Sun operating system.

6.1.13 Animation (runtime visualization)

Since LabVIEW is a dataflow language, the runtime visualization tracks the flow of data through the block diagram. It is a useful tool to determine which blocks are executing concurrently and which are executing sequentially. Also, the program may be placed into a step mode and the data tracked step by step.

6.1.14 Effective Use of Colors

LabVIEW is very effective at using colors to depict different data types and functions. The connections between blocks are color coded just as they might be in a physical electrical schematic diagram. This color coding eases the task of 'reading' the program.

6.1.15 Clarity of Graphical Symbols

For the intended audience, most of the graphical symbols are highly intuitive. By employing standard engineering symbols such as operational amplifiers and logic gates, LabVIEW's iconic structures are quite clear.

6.1.16 Interactive Capabilities

Front panel control settings can be modified during execution as well as x- and y-axis marker values for graphs and strip chart recordings. A block diagram cannot be modified during execution, nor will the program resume after correction of a runtime error from the time the error occurred (in other words, the program will be completely re-compiled and execution will begin at the beginning of the program).

6.1.17 Extensibility

Extending the functionality of a Virtual Instrument can be difficult. If a block is deleted from the block diagram, then all of its inputs and outputs are no longer valid. This causes the wires attaching the deleted block to other blocks to also become invalid. The inputs and outputs of the remaining blocks must then be rewired.

6.1.18 Interface Capabilities With Other Languages

Any data generated from a Virtual Instrument can be stored in standard format and ported to another software package for further analysis. Compiled C code can be imported into a LabVIEW VI, however, the compiled code must adhere to strict LabVIEW calling interface requirements resulting in a very cumbersome process. The Windows version does offer capabilities to interface to other Windows applications, but no testing was performed concerning this interface.

6.1.19 Analysis Capabilities

LabVIEW's data analysis capabilities are extensive. Some of the analysis available include: generating waveforms, determining frequency spectrum, determining power spectrum, applying digital windows and filters, separating real and imaginary components of an array, array mean, standard deviation, fitting waveforms both linearly and polynomial, dot and cross products of matrices, and solving linear equations. Figures 22 thorough 25 showed a snapshot of all analysis capabilities. This extensive library of analysis functions could easily convince any test system design engineer of the usefulness of LabVIEW applications. The following section will describe a VI which utilizes some of the analysis functions.

6.2 LabVIEW Implementation of a Spectrum Analyzer Virtual Instrument

LabVIEW was designed as an instrumentation software package taking advantage of the enhanced graphics capabilities of modern computer systems. In this area, LabVIEW's performance is quite impressive. This advanced performance is illustrated by the development of a Virtual Instrument which simulates a spectrum analyzer. The front panel and the diagram of the Spectrum Analyzer VI can be found in Figure 30 and 33, respectively.

Prior to detailing the development of the Spectrum Analyzer VI, a brief definition of spectrum analysis will be presented. "Periodic waveforms, regardless of shape, can be broken down mathematically into a series of sine waves." [Witte 1993]. Spectrum analysis is the process of determining the sine wave frequencies and amplitudes present in a signal, where frequency is defined as 1/(time for one cycle of a waveform) and amplitude is defined as the maximum height of a waveform. (For a mathematical treatment of spectrum analysis, please refer to Blackburn 1970.) The original signal is typically represented graphically with the y-axis as amplitude and the x-axis as time. A graph of Amplitude versus Time is referred to as a time domain graph or representation. Conversely, the representation of the spectrum analysis of a time domain signal, known as the spectrum, is a frequency domain graph. Frequency domain graphs consist of Amplitude versus Frequency data.

Figure 28 [Tektronix 1989] is a time domain representation of a square wave. This square wave remains at the maximum and minimum amplitudes for the same amount of time (in other words, the waveform has a 50% duty cycle). The period, or the time for one repetition of the waveform, is labeled as 'T' and the amplitude is labeled as 'A/2.' The fundamental frequency, f_1, of the waveform would therefore be calculated as: $f_1 = 1/T$.

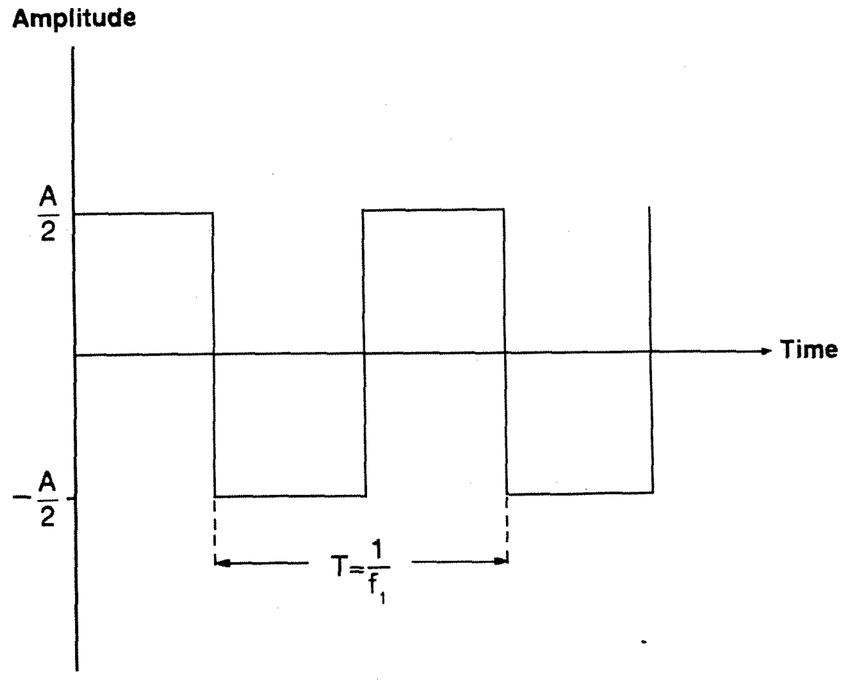


Figure 28. Time domain representation of a square wave with period = T, amplitude = A/2, and duty cycle = 50 %.

A square wave is composed of multiple sine waves with the frequencies of these sine waves being multiples of the fundamental frequency, f_1. A sine wave component whose frequency is a multiple of the fundamental frequency is referred to as a harmonic - the second harmonic is twice the frequency of the fundamental, the third harmonic is thrice the frequency of the fundamental, etc. The amplitude of the components of any rectangular waveform are limited by a

$$\frac{\mathrm{SIN}(\frac{\pi T}{2} f)}{(\frac{\pi T}{2} f)}$$

multiplicative function. A square wave is comprised only of the odd harmonics since

$$\frac{\mathrm{SIN}(\frac{\pi T}{2} f)}{(\frac{\pi T}{2} f)} = 0 \text{ when } f = nf_1, \text{ where } n = 2, 4, 6...$$

Figure 29 [Tektronix 1989] is a frequency domain representation of the square wave depicted in Figure 28. The odd harmonics occur at the peaks of the dashed sine waves. Theoretically, there are an infinite number of frequency components of a square wave.

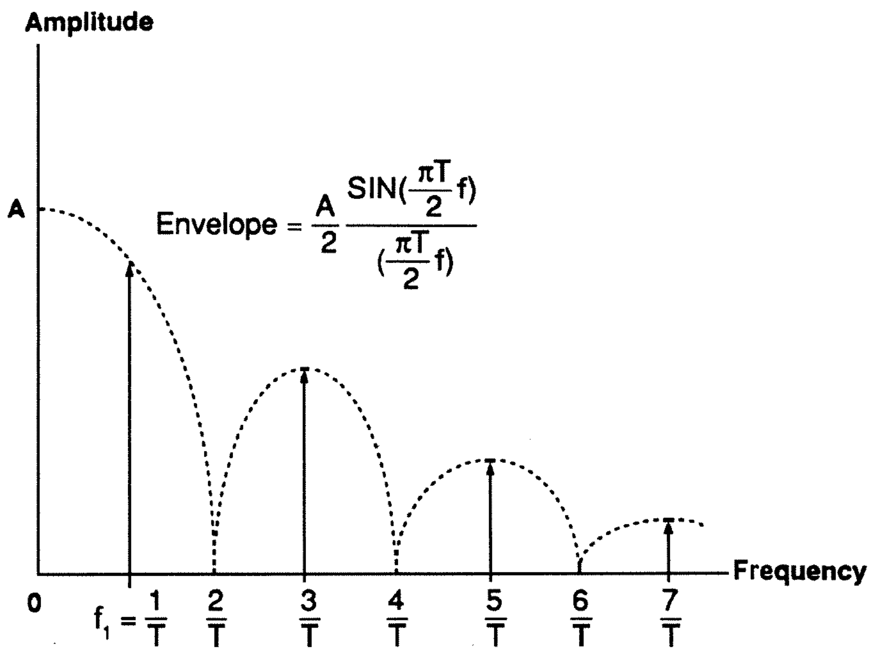
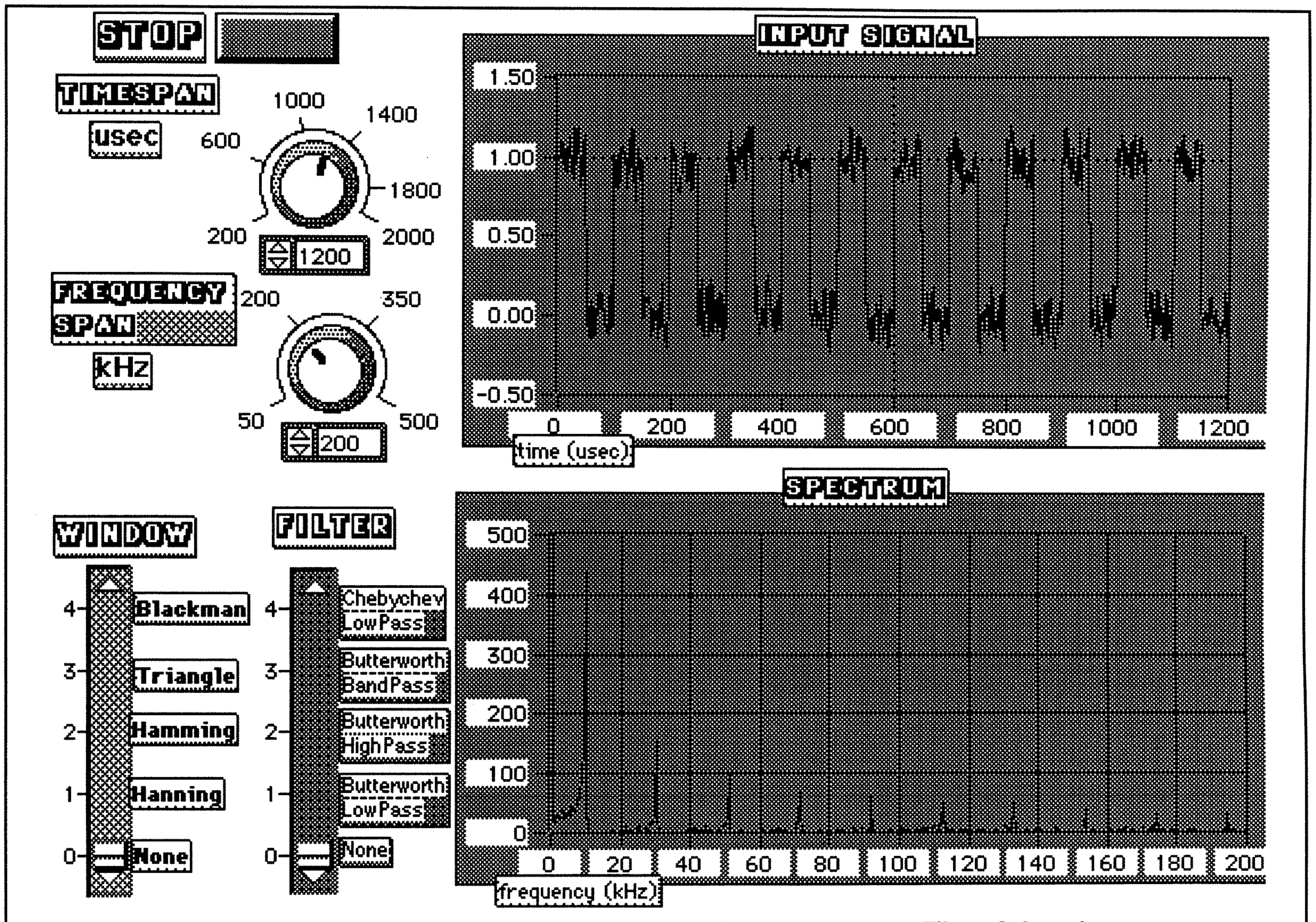


Figure 29. Frequency domain representation of a square wave with fundamental frequency f_1 .

With this brief definition of spectrum analysis, the LabVIEW Spectrum Analyzer VI can be explored. Figure 30 is a snapshot of the front panel. The Timespan and Frequency Span knobs control the x-axis values of the Input Signal (time domain) and the Spectrum (frequency domain), respectively. These knobs allow the user to choose the amount of the waveforms to be viewed. The Spectrum Analyzer VI also provides five choices of Windows or Filters to be applied to the input signal. The Windows 'clip' the ends of the input signal (see Figure 31). The Filters either remove higher frequencies from the input signal (Butterworth LowPass and Chebychev LowPass), remove lower frequencies from the input signal (Butterworth HighPass), or remove both very low and very high frequencies (Butterworth BandPass). Figure 32 illustrates the signals with the Butterworth LowPass filter chosen. When comparing the unfiltered input signal of Figure 30 with the filtered signal of Figure 32, it is apparent that the filtered signal has less noise, since the high frequencies have been removed (white noise is comprised of high frequency components).



TIMESPAN usec 1200

FREQUENCY SPAN kHz 200

Figure 30. Snapshot of Spectrum Analyzer VI With No Windows or Filters Selected.

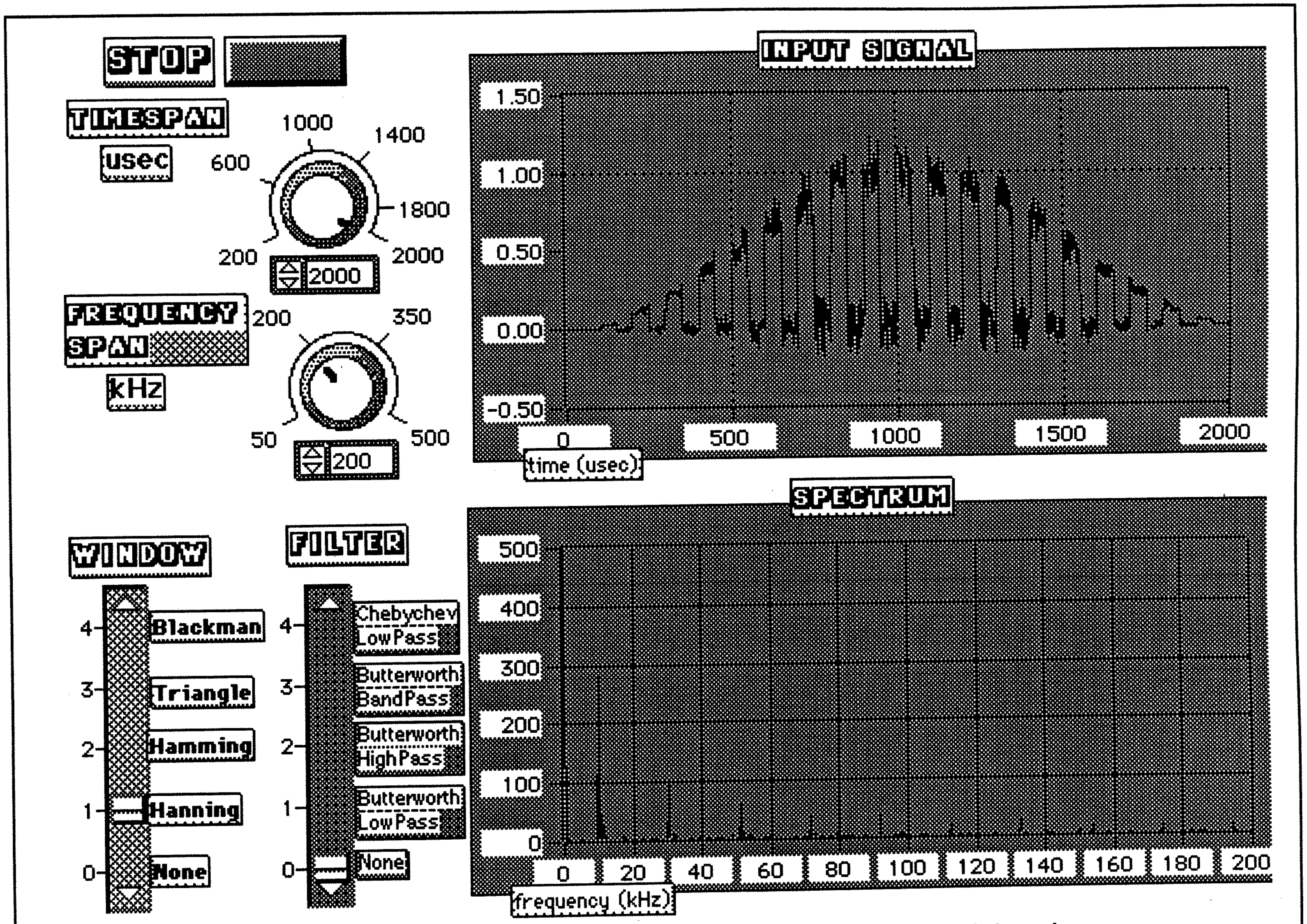


Figure 31. Snapshot of Spectrum Analyzer VI With Hanning Window Selected.

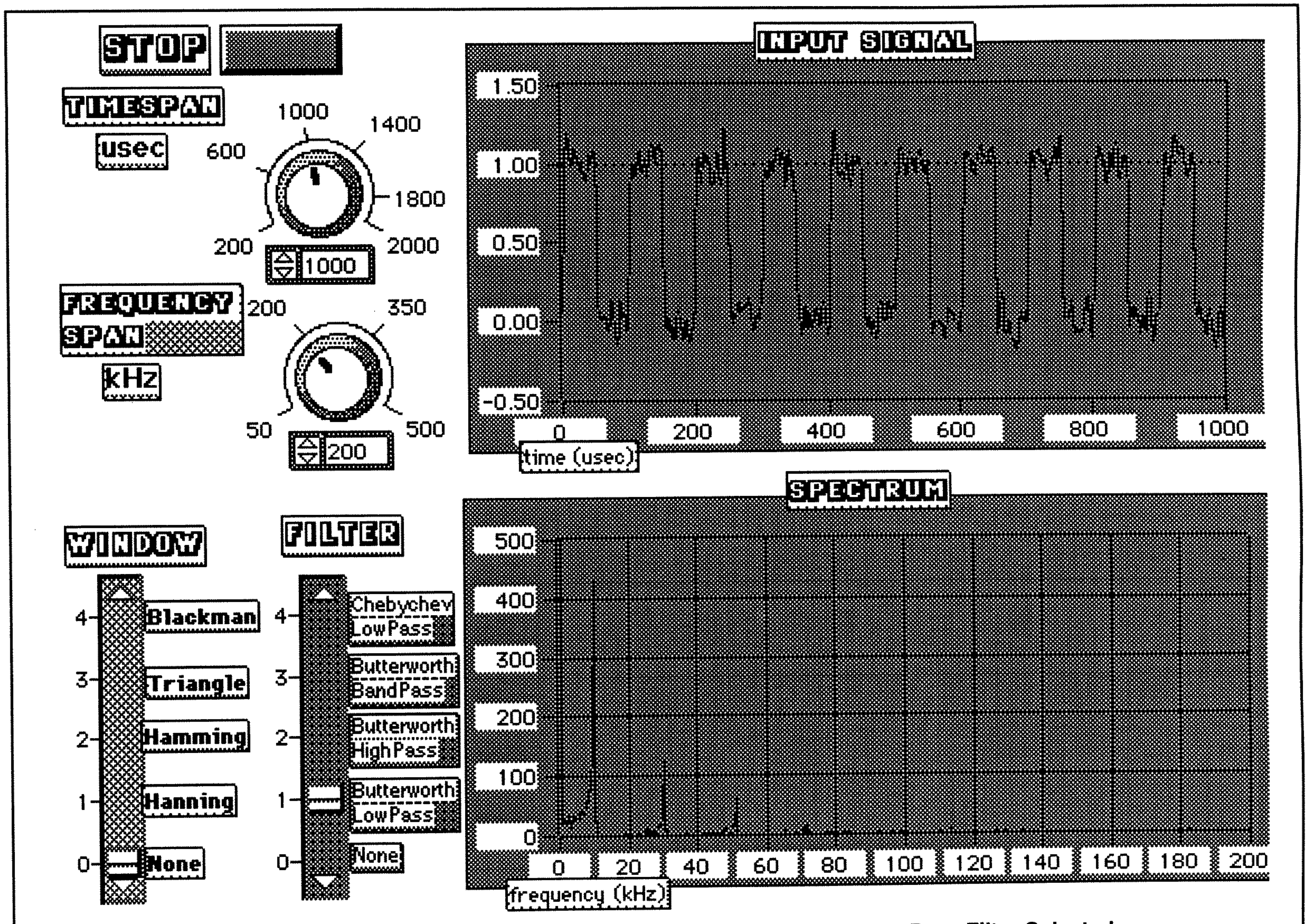


Figure 32. Snapshot of Spectrum Analyzer VI With Butterworth Low Pass Filter Selected.

The Spectrum graph represents the frequency domain of the Input Signal waveform. The fundamental frequency of the Input Signal is 10000 Hz (1/100 μsec). At each odd harmonic, there are spikes on the Spectrum waveform representing those components. Since the Input Signal is not a perfect square wave, the spectrum is not simply composed of perfect spikes at each of the odd harmonics. The induced noise creates additional frequency components.

Figures 33, 34 and 35 illustrate the LabVIEW program. As can be seen, all of the program is enclosed in a while loop, which will continue to execute until the STOP button is chosen on the front panel. When reading the while loop from left to right, the first operations to be performed are creating a square wave and an array of white noise. The noise is then filtered and added to the square wave and placed in the array labeled SIGNAL. The signal is then processed through the case block determining which window is to be used. The resultant signal is then processed through a second case structure determining the chosen filter. This resultant signal is then routed to be displayed as the input signal and to the block which converts the signal to its frequency components and displayed as the spectrum.

The block which converts the time domain signal into the frequency domain, $\mathfrak{F}(x)$, performs a Fast Fourier Transform (FFT) on the input array. The FFT is a discrete and efficient implementation of the Fourier integral. The FFT is calculated as follows [National Instruments 1990]:

$$Y[i] = \sum_{k=0}^{n-1} X[k]\, e^{(-j2\,\Pi\, ik/n)} \quad \text{, for } i = 0, 1, \ldots, n-1$$

where Y[i] is the i^{th} element of the FFT of X and j is sqrt(-1). Direct implementation of this equation requires approximately n^2 operations, however, if the size of the input array is limited to a power of 2, a significant number of operations can be eliminated and a fast algorithm can be implemented. The algorithm implemented for $\mathfrak{F}\{X\}$ in the LabVIEW Analysis VI Library is known as the Split-Radix algorithm. This algorithm has a form similar to the Radix-4 algorithms with the efficiency of the Radix-8 algorithms. The Split-Radix algorithm requires the fewest number of multiplications over the Radix-2, Radix-4, and Mixed-Radix algorithms [National Instruments 1991].

The use of the $\mathfrak{F}\{X\}$ block results in an array with both the positive and negative harmonics of the input signal. Only the first half of the array (the positive harmonics) are utilized in the VI. $\mathfrak{F}\{X\}$ also produces both real and imaginary values of the frequency components, but only the real components are required for the VI.

The array of real and positive frequency components of $\mathfrak{F}\{X\}$ are then converted to a graph and displayed on the front panel Spectrum display.

This VI could easily be modified to incorporate either an IEEE-488 data acquisition instrument or a plug-in Analog to Digital Converter board. Rather than generating a square wave from the LabVIEW Analysis menu, an actual signal could be captured and then processed through the $\mathfrak{F}\{X\}$ analysis block. It is especially convenient that both the input and resulting spectrum signals can be viewed on the same display. The test system designer is then not limited to placing the data acquisition in full view of the test system operator. In this way, the designer can limit access to instrument control settings by determining settings through the LabVIEW front panel only.

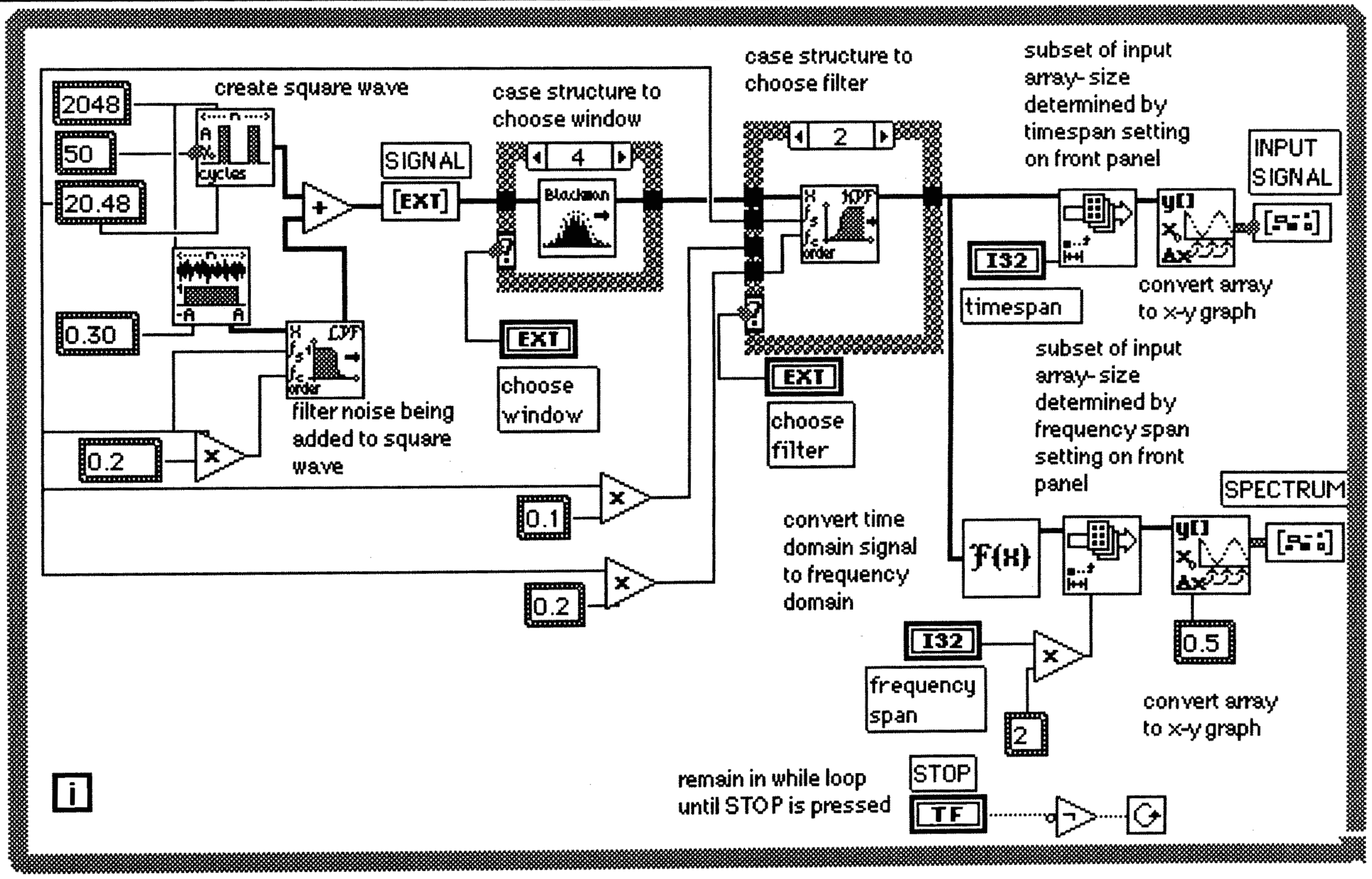

2048
50
20.48
create square wave
A
cycles
0.30
-A
A
LPF
filter noise being added to square wave
0.2
SIGNAL
[EXT]
case structure to choose window
4
Blackman
EXT
choose window
0.1
0.2
case structure to choose filter
2
HPF
order
EXT
choose filter
subset of input array- size determined by timespan setting on front panel
I32
timespan
INPUT SIGNAL
convert array to x-y graph
subset of input array- size determined by frequency span setting on front panel
SPECTRUM
convert time domain signal to frequency domain
F(x)
I32
frequency span
2
0.5
convert array to x-y graph
remain in while loop until STOP is pressed
STOP
TF
i

Figure 33. Spectrum Analyzer Virtual Instrument LabVIEW Program.

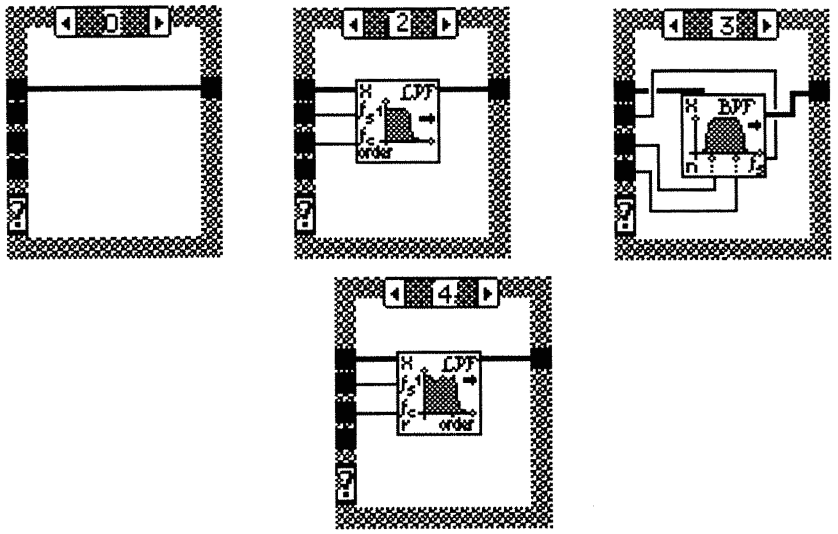


Figure 34. Additional choices for Filter Case Structure in Spectrum Analyzer LabVIEW program.

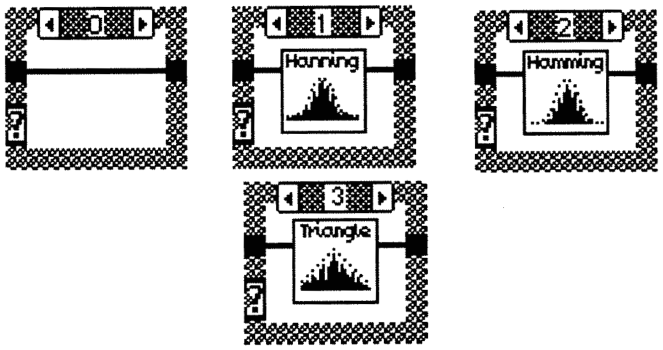


Figure 35. Additional choices for Window Case Structure in Spectrum Analyzer LabVIEW program.

6.3 ProGraph

6.3.1 Overview

ProGraph provides a strong environment for Macintosh application development. The ProGraph development system is based upon a three tier level as illustrated in Figure 36 [TGS 1992B]. On the first tier is the ProGraph language itself, composed of a visual, object-oriented dataflow language. Applications are built using the traditional Macintosh windows and pull-down menu structure. The second tier encompasses the manner in which windows and menus are designed and how events are handled by utilizing ProGraph system classes. Finally, the third tier is composed of the editor, interpreter, and compiler. The editor is designed as a traditional Macintosh graphical user interface with 'point and click' features and pull down menus. The editor also provides extensive on-line help. ProGraph offers an advanced interpreter which allows for stepping, tracing, debugging, and modification of the program during execution. Once the program has been completed, it can be compiled as a stand-alone Macintosh application, increasing execution speed and reducing memory requirements over interpreted programs therefore eliminating the need for an interpreter.

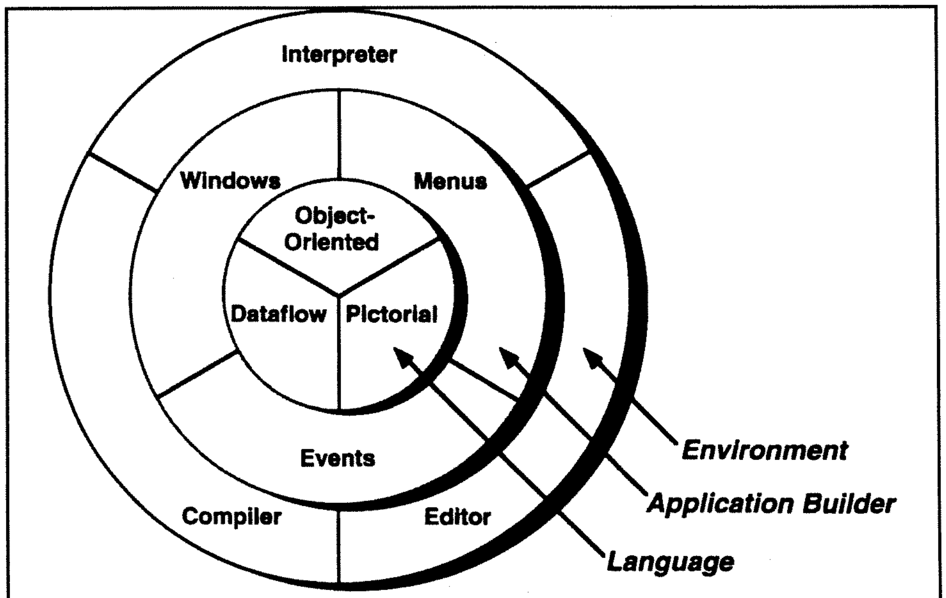


Figure 36. Levels of the ProGraph Development System [TGS 1992B].

A ProGraph program consists of [TGS 1992A]:

- classes (with their associated attributes and methods)
- universal methods
- operations (user-defined and system-supplied, with associated controls)
- data objects (instances of classes and primitive data types)
- persistents (container objects)

Classes and methods will be discussed in greater detail in the **Paradigm** criterion below. "An operation is the basic executable component of a method. It has a name, zero or more inputs, zero or more outputs, and a distinctive icon. It can operate on input data and it can produce output data; it may also produce side effects (that is, beyond producing output data it may also change the state of an object, such as that of a window on the screen)." [TGS 1992A]. Operations can either be user defined operations or ProGraph operations. The Operations pop-up menu can be found in Figure 37. Since the meaning of Operations is not immediately obvious, a few of the operations will be introduced. A *Simple* operation can call a primitive (a system-supplied compiled operation), a Macintosh Toolbox routine, or a class-based or universal method. A snapshot of sample available primitives can be found in Figure 38. A *Persistent* operation accesses the value of a persistent (container object) whose name appears within the operation icon. A new instance of a class is created by executing the *Instance* operation with the name of the desired class within the icon. The *Get* and *Set* operations access and set the value of the attribute listed, respectively. A *Local* operation is an encapsulation of a body of code into a single icon. Using a local operation conserves screen space since a group of operations can be made into a single icon (that is, a local operation) by selecting the *Opers to Local* operation.

Data objects flow through the program (i.e. dataflow programming). Data objects are not limited to simple predefined data types. Data objects can themselves be instances of a class. The ability to create such complicated data objects is an impressive strength of ProGraph.

As was previously mentioned, persistents are container objects. They are named elements that can hold any value. This value is retained between executions of a program and is also saved by the ProGraph editor along with its program. Figure 39 illustrates the icon associated with a Persistent.

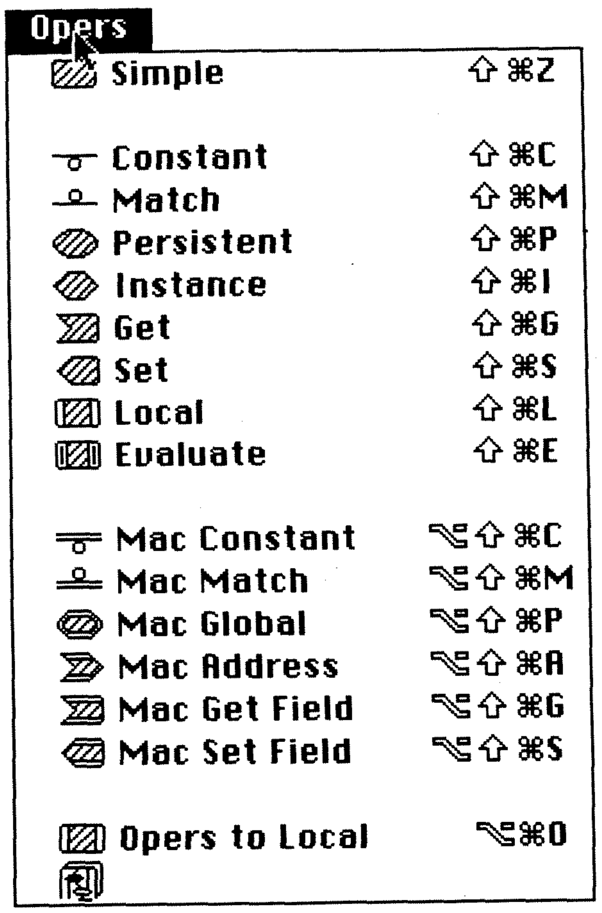
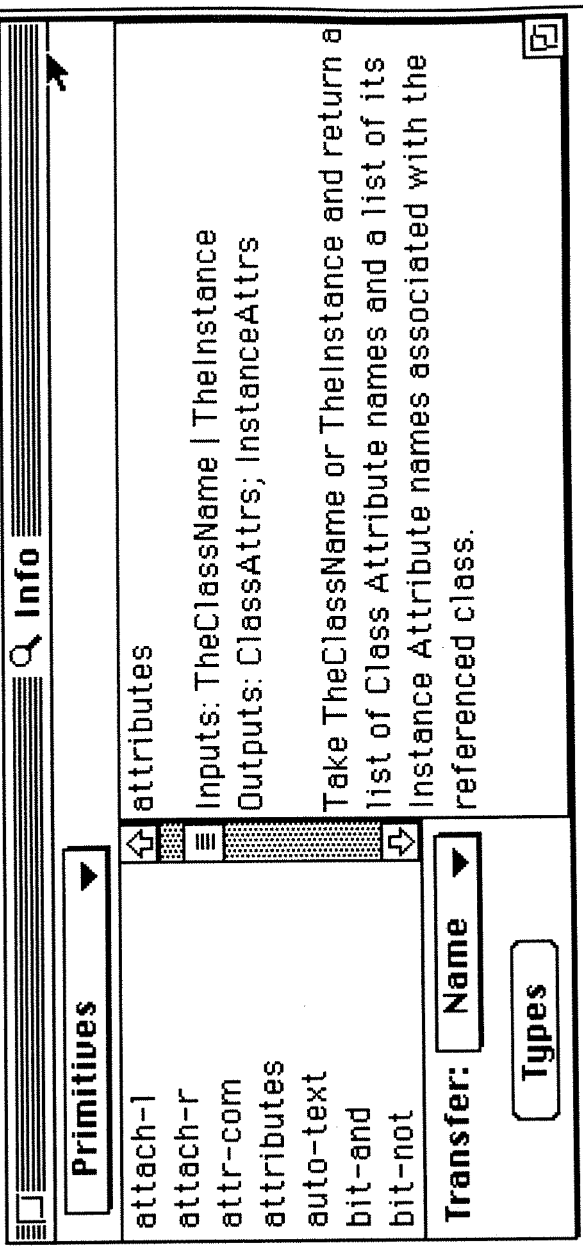


Figure 37. ProGraph's Operations Menu.



attach-l
attach-r
attr-com
attributes
auto-text
bit-and
bit-not

attributes

Inputs: TheClassName | TheInstance
Outputs: ClassAttrs; InstanceAttrs

Take TheClassName or TheInstance and return a list of Class Attribute names and a list of its Instance Attribute names associated with the referenced class.

Figure 38. Sample ProGraph Primitives and Accompanying Description.

Figure 39. ProGraph Icon Representing a Persistent (container) Object.

6.3.2 Scope

ProGraph has been designed as a general purpose object-oriented programming language. This does not intimate that ProGraph is a simple language. On the contrary, it is a very high level language as was depicted on Shu's three dimensional evaluation presented previously. Essentially any system which can be designed as an object-oriented design can be implemented using ProGraph. Its limitations are inherent with any visual programming language - the ability to convey functionality within the available screen space.

6.3.3 Intended Audience

ProGraph is intended for the general programming audience familiar with the concepts of object-oriented design and dataflow computing. Although programs can be developed which are not object-oriented, significant advantages of ProGraph would be lost. ProGraph was intentionally designed to minimize the influence of natural language thereby including a broader audience.

6.3.4 Paradigm

ProGraph is based upon *object-oriented dataflow programming.* As with any dataflow language, each node begins execution only when data is available at all of its inputs. This paradigm allows for creation of diagrams with independent or parallel dataflow paths and simultaneous operation.

Although ProGraph is a dataflow programming language, instructions can be forced to execute in a specific order if so desired. The structure to order execution is referred to as a *synchro.*

6.3.4.1 Object-Oriented Programming

For a language to be considered as an object-oriented language, it must conform to three essential principles: classification of objects, encapsulation, and inheritance [Fichman and Kemerer 1992]. Stroustrup describes the object-oriented paradigm as: "Decide which classes you want; provide a full set of operations for each class; make commonality explicit by using inheritance." [Stroustrup 1988]. Any language which does not possess these qualities is not defined as object-oriented but perhaps as an object-based or a data abstraction language.

6.3.4.2 Classification of Objects and Encapsulation

An object is a logical collection of data and associated methods. The inclusion of data and methods within one entity is known as encapsulation. Encapsulation also implies that objects possess a private data store accessible only from within the object itself and a public interface accessible by other objects. Objects are then grouped together in classes. A class is an abstract description or template of a particular object type that describes data and methods to be associated with objects of that class. The description of the data of a class is comprised of the name and type of the attributes.

The ProGraph icons associated with classes, methods, attributes, and inherited attributes are presented in Figure 40. (An explanation of inheritance and inherited attributes will follow). As can be seen, a class is comprised of attributes (left side of class icon) and methods (right side of class icon). Methods are a sequence of operations connected by data links. Inherited attributes include an arrow indicating that the attribute was inherited from a parent class.

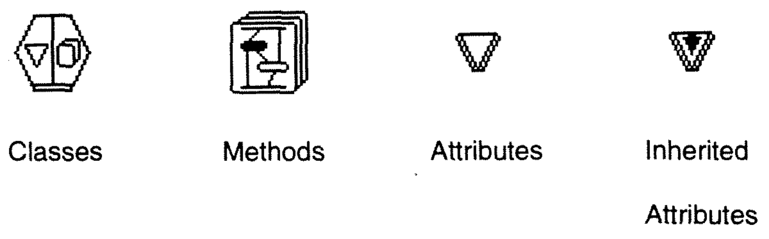


Figure 40. ProGraph Icons For Classes, Methods, Attributes, and Inherited Attributes.

6.3.4.3 Inheritance

Inheritance is defined as one class obtaining all the attributes and the full library of methods specified in another class. Furthermore, inheritance can be divided into single and multiple inheritance. Single inheritance specifies that a class inherits from only one class as opposed to multiple inheritance, where a class can inherit from more than one parent class. For example, Figure 41 illustrates the single inheritance of ProGraph's system classes. The *Window Item* class is a subclass of the *System* class, therefore, *Window Item* inherits all of *System's* attributes and methods. The subclass, *Window Item* can add its own unique attributes and methods to those inherited from its parent class *System*. Figure 42 displays the attributes of the *System* and *Window Item* classes. The attributes inherited by *Window Item* from *System* are denoted by an arrow in the icon (the attributes named owner, and FALSE). Attributes added within *Window Item* are then inherited by its subclasses.

Methods are also inherited from a parent class. A subclass can add methods or it can overshadow methods already defined in the parent class. Overshadowing or overloading a method is defined as polymorphism, where one syntactic object means more than one thing. Figures 43 and 44 list the methods of Window Item and Scroll List. Scroll List adds two new methods, namely Key and Tab To Item. The method Mouse Down is overshadowed by the new definition in Scroll List. Figure 45 displays the Mouse Down method of class Scroll List. On the right side of the figure, there is a simple operation labeled //Mouse Down. The arrow in the icon is referred to as a *Super* annotation. The *Super* annotation tells ProGraph to look for the method Mouse Down, not in the Scroll List class but in the Window Item class. If ProGraph does not find the method in the parent class of the current class, it continues up the inheritance line until it finds a method by the indicated name. In this manner, a class can exhibit the behavior of its predecessor and add special functionality.

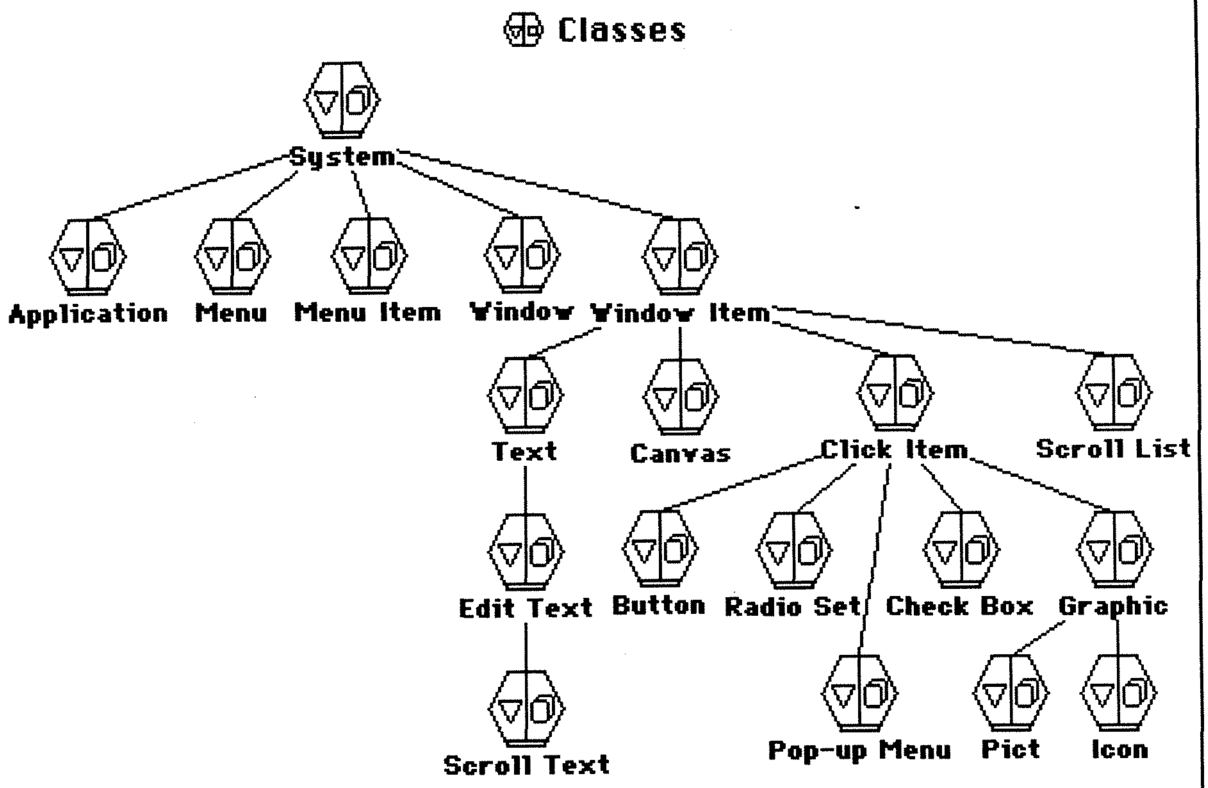
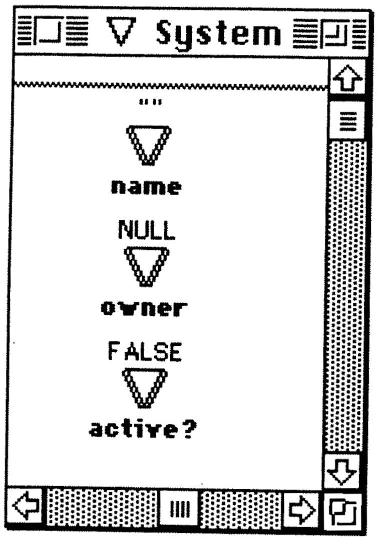


Figure 41. ProGraph's System Classes With Single Inheritance.



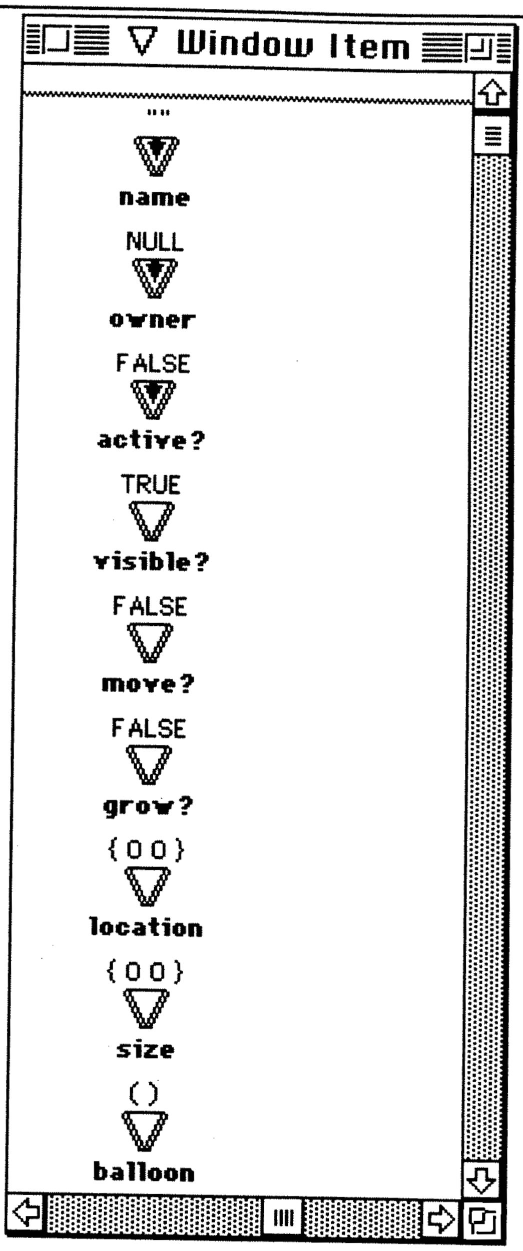


Figure 42. System Class Attributes and Window Item Attributes (both inherited attributes and new attributes)

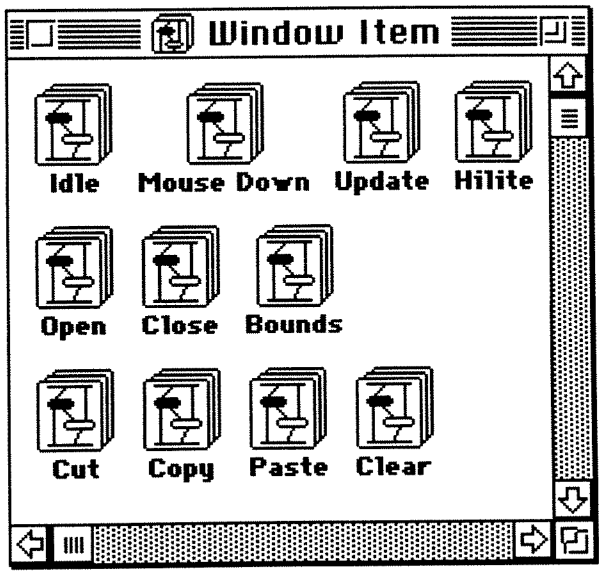


Figure 43. Window Item Methods.

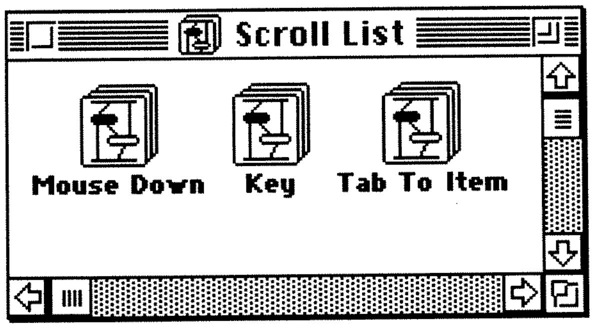


Figure 44. Scroll List Methods. Mouse Down Method Overshadows Mouse Down Method From Parent Class.

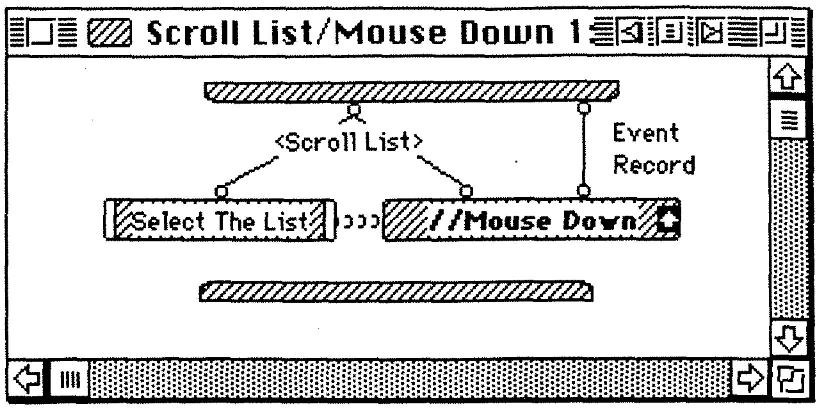


Figure 45. Mouse Down Method of Scroll List Utilizes Mouse Down Method of Parent Class.

6.3.5 Ease of Use

Object-oriented concepts are presented quite clearly in ProGraph. The inheritance mechanism of ProGraph is easily understood with the graphical representation of class hierarchy. A full list of attributes can be obtained by double-clicking on the left side of a class icon with inherited attributes distinguished from a class' defined attributes. When double-clicking on the right side of a class icon, only the methods defined (or re-defined) within that class are displayed. To determine the methods inherited, one must double-click on the ancestors' class icons.

The concept of building methods from operations is self-explanatory, but the functionality of operations and primitives are not readily explained by the icons or the names of the icons. Without strong guidance from the Reference and Tutorial Manuals, it would be extremely difficult to implement a design.

If a programmer is experienced in writing Macintosh applications, then building an application in ProGraph can be considered a fairly simple task. A novice to the world of Macintosh application design would initially find the process confusing and tedious. Building an application with even the most rudimentary menuing schemes can be a daunting task when first introduced to ProGraph. As experience with ProGraph is gained, application implementation does become less cumbersome.

ProGraph does have extensive on-line help capabilities which facilitates the ease of use.

6.3.6 Visual Representation

ProGraph is a simple iconic visual language with textual annotation. Most icons are variations of a rectangular shape. This simplicity can make it difficult to distinguish between different types of icons. The iconic representation for classes and methods are more complex and are more easily recognized.

6.3.7 Compiler

While developing the system, the program is graphically interpreted (as opposed to compiled) for ease of debugging. After final debugging, a program can then be graphically compiled into machine code for faster execution.

6.3.8 Reusability

Object-oriented designs offer a great range of flexibility through the modification of inherited attributes and methods. This flexibility lends itself to reusability. If a class can be added to an existing design with only minor modifications to inherited attributes and methods and the addition of new attributes and methods, then the case for reusability is strong. Of course, simply because a language is object-oriented does not guarantee that code will be developed in a reusable manner.

6.3.9 Data Structures and Types

ProGraph offers a wide range of system defined types. In addition to these system defined types, classes themselves can be considered data structures. An instance of a class can be passed as a data object along the data links between operations. With this flexibility, the complexity of data structures is essentially unlimited.

6.3.10 Effective Use of Screen Area

Within ProGraph, any portion of a method can be reduced to an icon for a local method to conserve screen space. This new local method can then be opened separately to determine its function.

6.3.11 Hardware

ProGraph is currently available only for the Macintosh hardware platform.

6.3.12 Operating Systems

ProGraph is currently available to run only under the Macintosh operating system.

6.3.13 Animation (runtime visualization)

ProGraph's interpreter is highly advanced. Execution of an application can be followed by tracking the data as it moves from one operation to the next. In addition, the stack can be dynamically displayed to determine the state of the system at any time.

6.3.14 Effective Use of Colors

ProGraph is limited in its use of color. During execution, data can be tracked by watching the changing colors of the operations. Also, when an error is encountered, the screen will take on a new color to indicate that an error has occurred.

6.3.15 Clarity of Graphical Symbols

Without annotation, most of the graphical symbols would not be easily recognized. After experience with ProGraph, the symbols become more familiar.

6.3.16 Interactive Capabilities

The interactive capabilities of ProGraph are very strong. Since the program is interpreted, parts of the application can be developed 'on the fly' as the interpreter realizes that a called method does not exist. The program will resume running from the point when the error occurred. This interpretive capability greatly speeds development time.

6.3.17 Extensibility

By virtue of its objected-oriented paradigm, ProGraph applications can be easily extended by adding new classes and methods. If an application has been designed in an effective object-oriented manner, then adding classes to the hierarchy or independent classes has little effect on existing classes. Since ProGraph allows polymorphism, altering inherited methods allows for great flexibility and function.

6.3.18 Interface Capabilities with Other Languages

Both the ProGraph interpreter and the ProGraph compiler allow C code to be imported. "There are two different formats for writing imported C code. One is for writing external primitives, which can be included in the ProGraph environment. The other format is used for writing external code which is linked by the compiler into the final application." [TGS 1992B].

6.3.19 Analysis Capabilities

ProGraph is very limited in its analysis capabilities. It does provide simple trigonometric functions and information on class hierarchy and attributes. It was not designed for data analysis.

6.4 ProGraph Implementation of an Object-Oriented Gradebook Application

ProGraph has been designed as an object-oriented, dataflow, visual language. As an object-oriented language, class structures, encapsulation, and inheritance are inherent attributes of ProGraph. ProGraph is also designed to build Macintosh applications complete with pull-down menus, point-and-click features, and pop-up windows. To illustrate the object-oriented paradigm and Macintosh application design, the development of a Gradebook Application will be presented.

The Class Hierarchy of the Gradebook application is displayed in Figure 46. The System Classes introduced earlier must be included in the application to develop the Macintosh interface. The two classes created for Gradebook are Person and Student. Figure 47 lists the attributes of both classes, with the attributes Name and Age containing arrows in the Student Class indicating that these attributes were inherited from Person. The Student class adds the attributes Grades, Grade1, Grade2. Grade3, Grade4, and Average. Each instance of the class Student will possess all of these attributes. The Student class also includes a Class Attribute, NumGrades, denoted by the hexagonal shape ⬡. A Class Attribute is "owned" by the class and is known by and accessible by all instances of the class.

The Gradebook Application contains three Universal Methods (methods not belonging to a class but accessible throughout the design). These methods presented in Figure 48 are: Initial, used in the creation of windows; Sort Names, which returns a sorted list; and Average, which returns an average of a list. The Gradebook application also includes one container object or persistent, Gradebook, as shown in Figure 48. This persistent contains a list of Student instances (see Figure 49). The attributes of the individual instances can be examined by double-clicking on the instance icon. Figure 50 lists the attributes of Student instance 4. By double-clicking on the attribute icons of the Student instance, a dialog box is produced allowing the user to modify that attribute.

The methods contained in the class Student are presented in Figure 51. All of these methods are accessible from any instance of class Student. The combination of attributes and methods within class Student is an illustration of encapsulation. For brevity, only a few of the methods in the Student class will be presented in detail. A complete listing of all the Student methods can be found in Appendix A.

The method Student/Add first calls a local method which creates an instance of student with the Name attribute set (see Figure 52). This new instance is then added to the persistent Gradebook by invoking the attach-r primitive. The scroll list located in the User window is then updated with the new Gradebook persistent. An example of the User window is illustrated in Figure 53. The user enters the last and first names of the new student. When the button labeled 'Add Student' is clicked, the method Student/Add is invoked, causing the student to be added to the Gradebook persistent and updating the scroll list of students located in the center of Figure 53.

Figure 54 is the diagram of method Student/ClassAverage which is invoked when the 'Class Average' button in the User window is clicked. The Grade1, Grade2, Grade3, and Grade4 attributes of all the instances contained in the Gradebook persistent are accessed (the ellipses denotes that a list has been processed). The Universal Method, Average, is then invoked to determine the class average. Finally, the show primitive is used to display a dialog box indicating the class average (see Figure 55).

When a new student is added to the Gradebook, the values for the attributes Grade1 through Grade4 are NULL. A user enters the student's grades by first highlighting the student's name in the User window scroll list and then clicking the button labeled 'Enter Student's Grades.' The method Student/EnterGrade is used to set the Grade1 through Grade4 attributes of the Student instance (see Figure 56). The user is forced to enter each grade in order. This forced sequence is accomplished through the use of synchros (represented as rows of semicircles in Figure 56).

As the diagram shows, Grade1 is entered before Grade2, as with Grade2 before Grade3 and Grade3 before Grade4. The grades are entered through a dialog box as is illustrated in Figure 57.

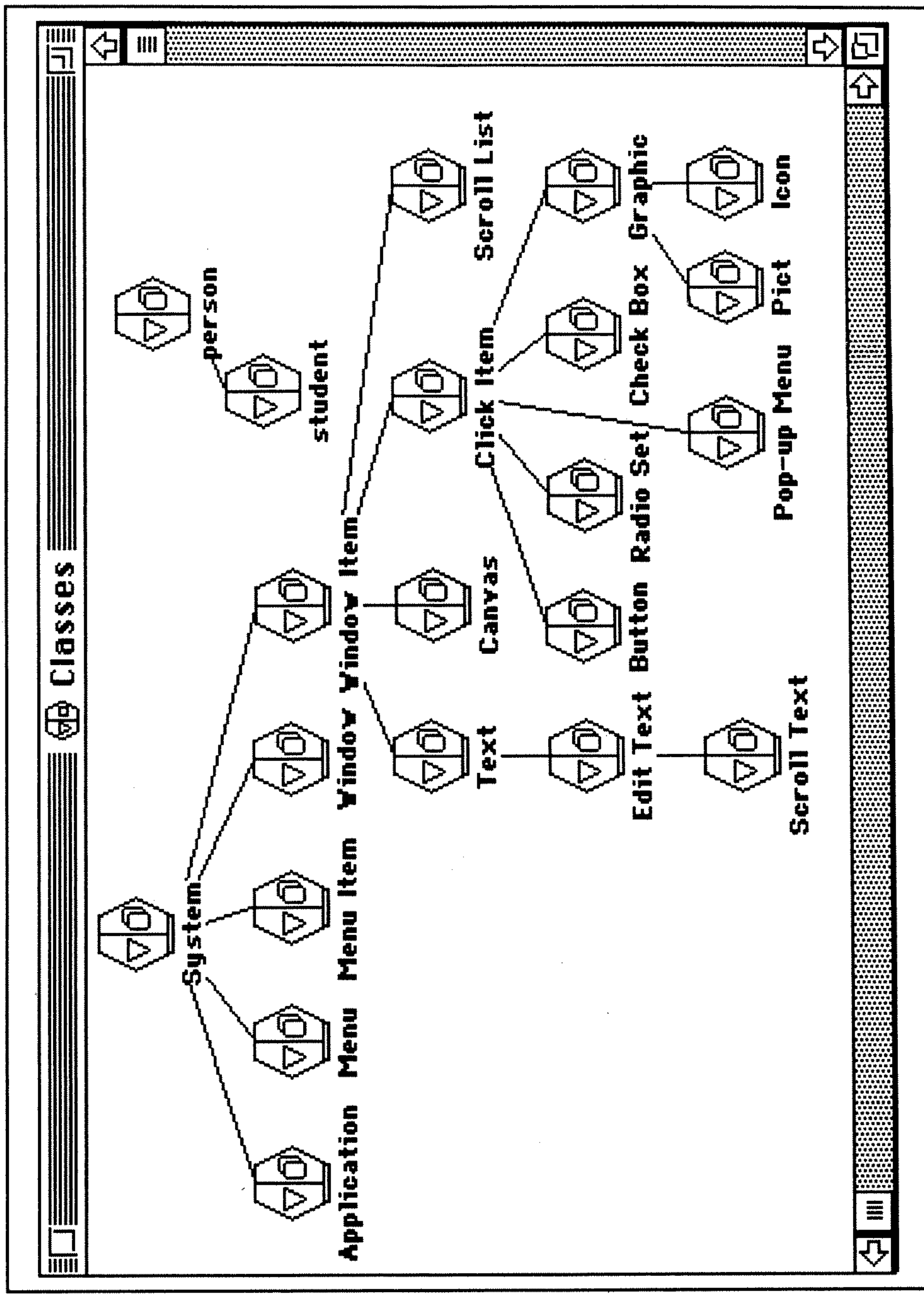


Figure 46. Class Hierarchy for ProGraph Gradebook Application.

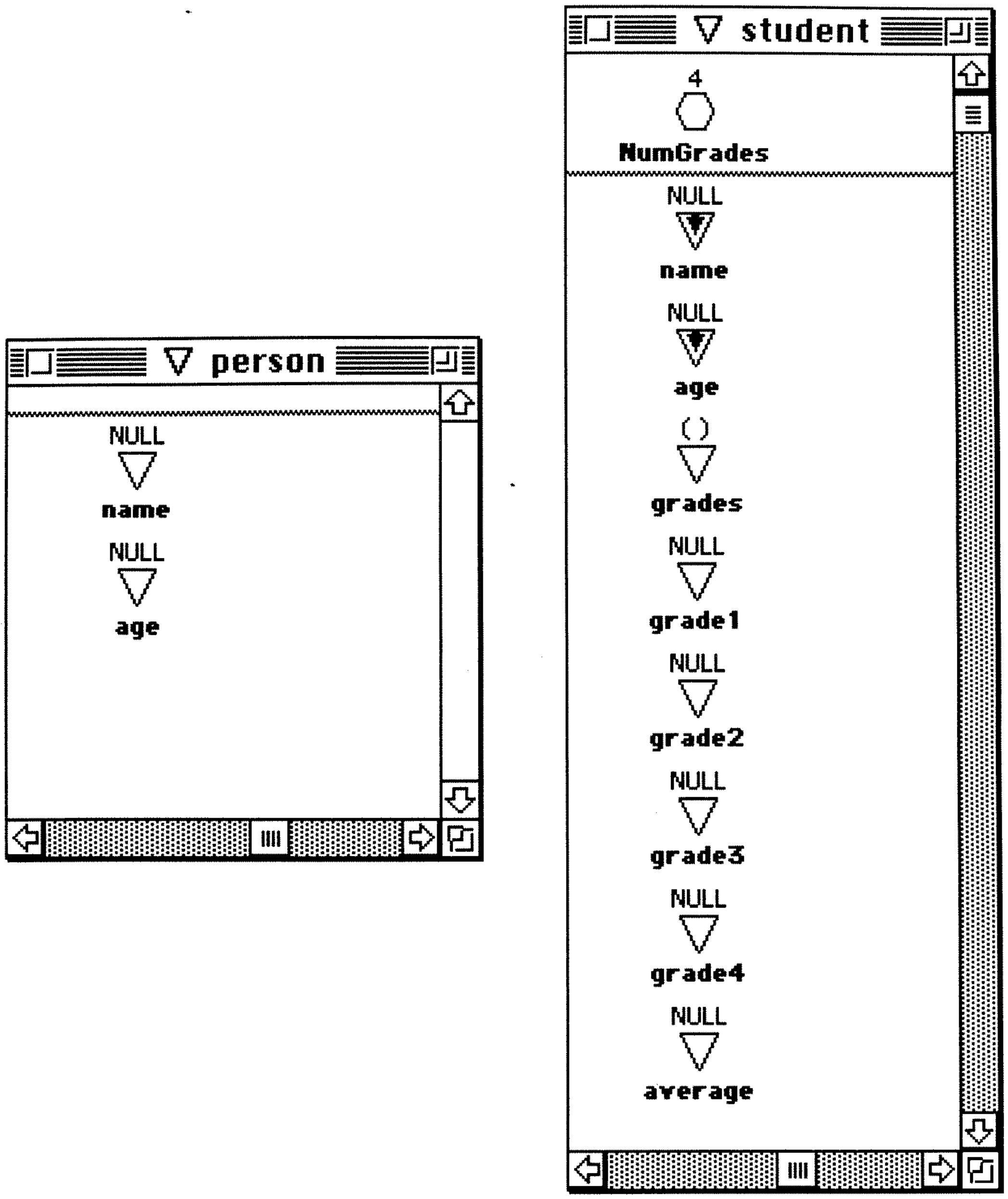


Figure 47. Attributes of Person and Student Classes.

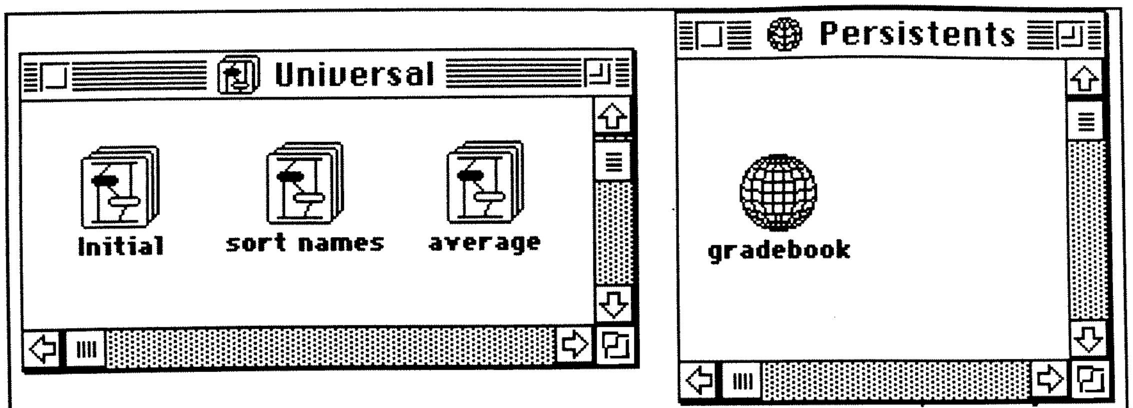


Figure 48. Universal Methods and Persistents of ProGraph Gradebook Application.

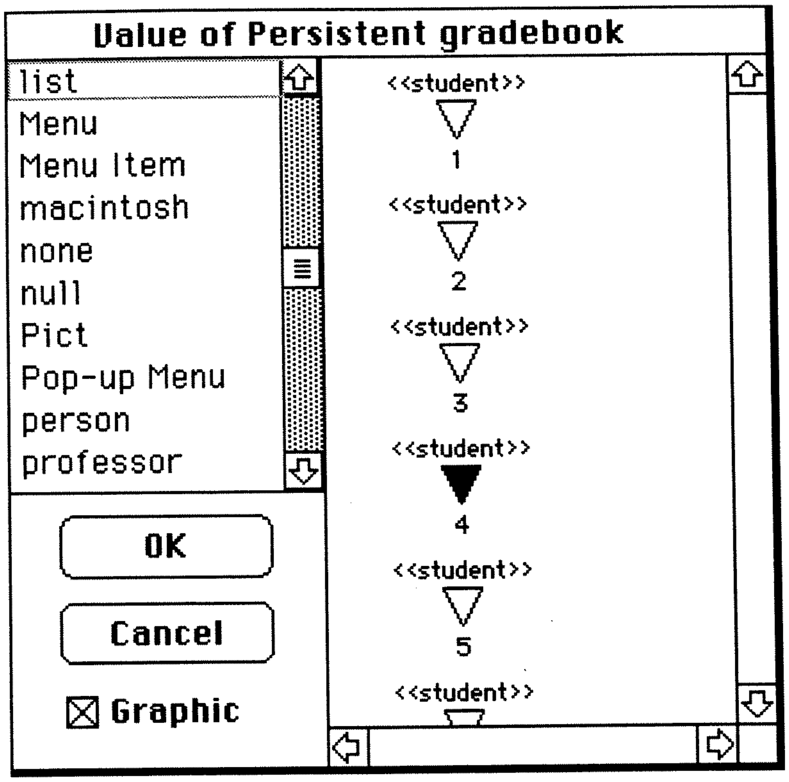


Figure 49. Values Contained In the Persistent Gradebook (container object).

The scroll list located in the upper left corner indicates the possible data types for the values contained in the Persistent Gradebook. The values in Persistent Gradebook are of type *list*.

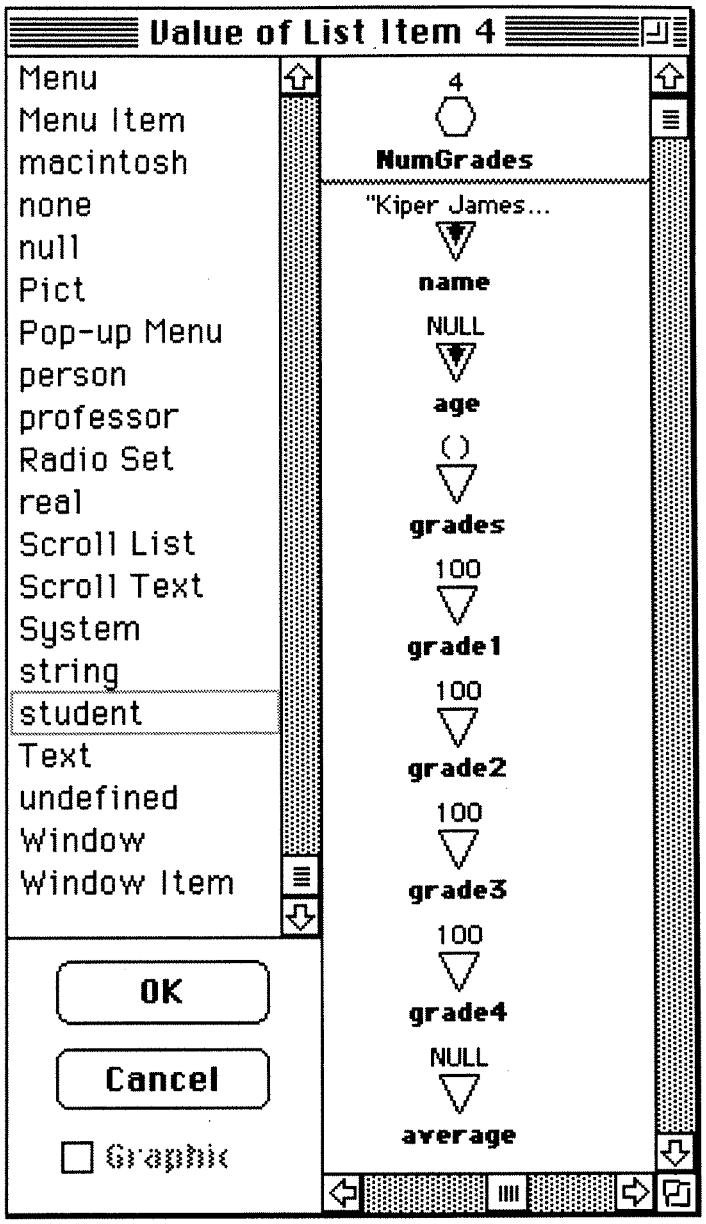


Figure 50. Values of Student Instance 4 Located in Persistent Gradebook.

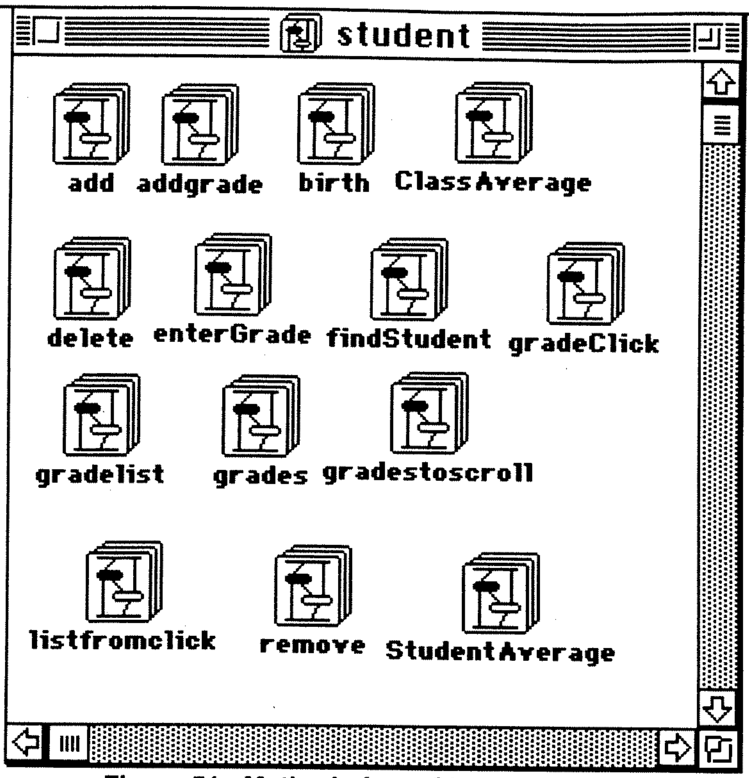


Figure 51. Methods from Class Student of ProGraph Gradebook Application.

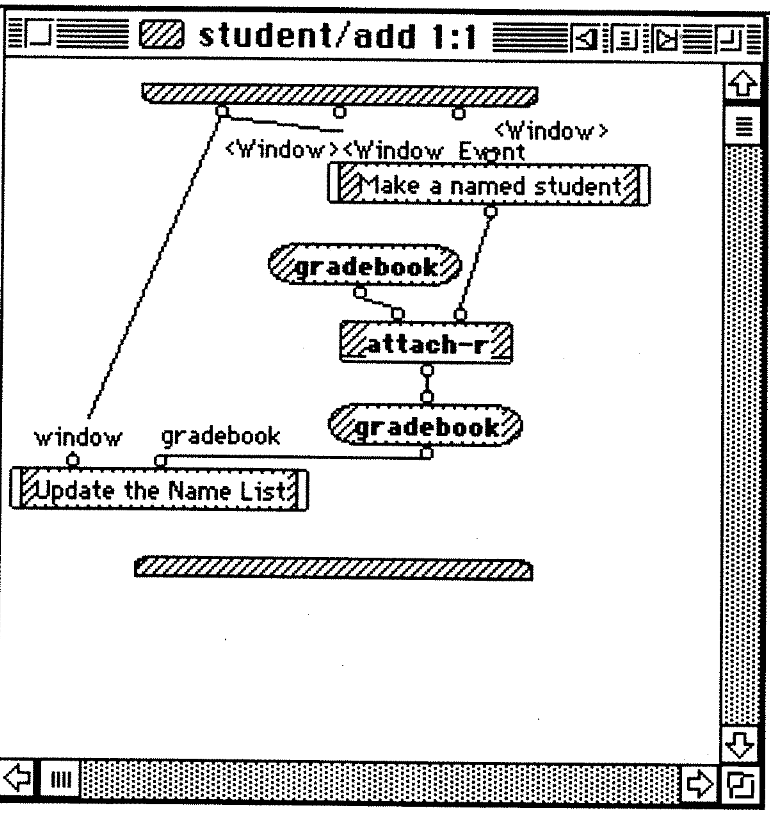
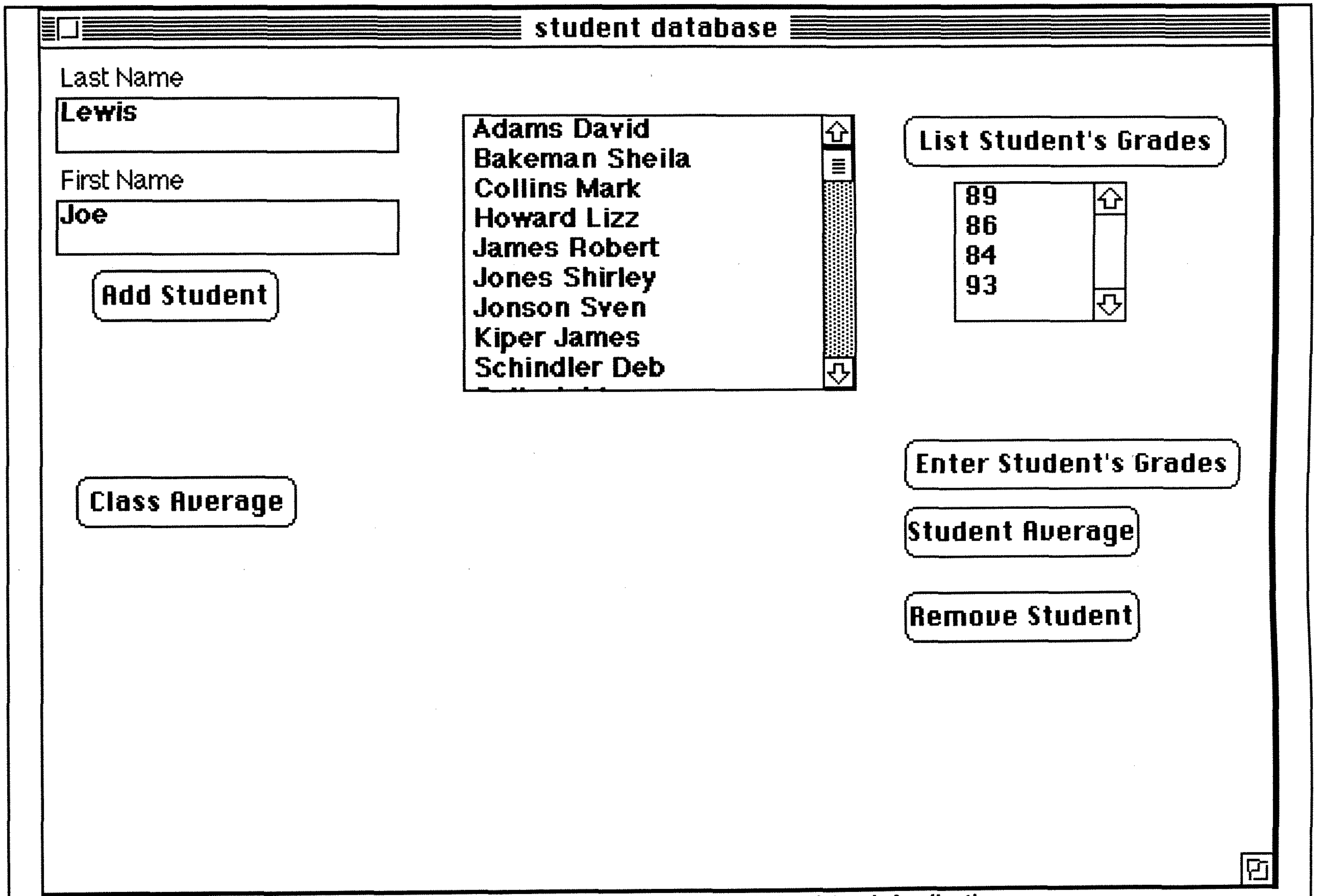


Figure 52. Method Add of Class Student.



Last Name
Lewis

First Name
Joe

Adams David
Bakeman Sheila
Collins Mark
Howard Lizz
James Robert
Jones Shirley
Jonson Sven
Kiper James
Schindler Deb

89
86
84
93

Figure 53. Sample User Window of ProGraph Gradebook Application.

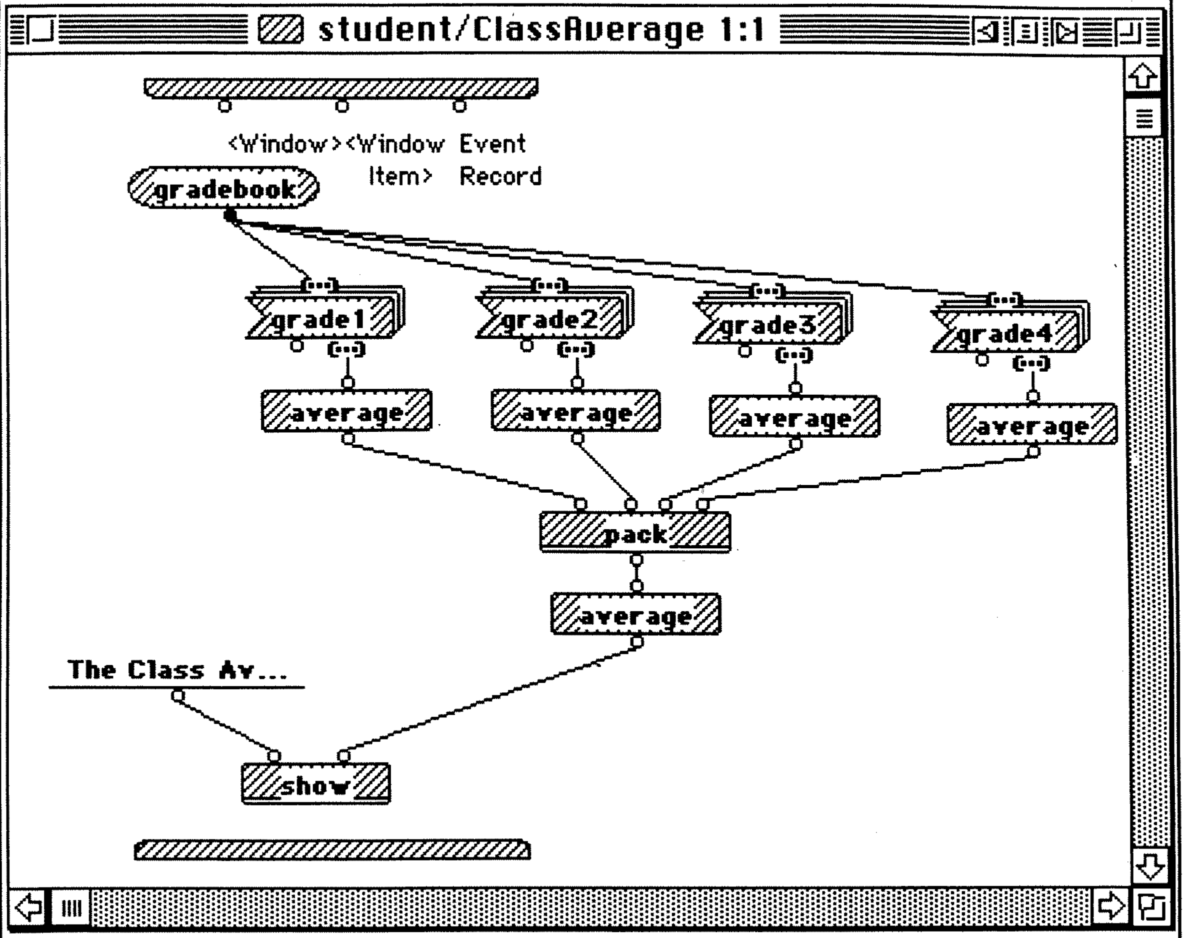


Figure 54. Method ClassAverage of Class Student.

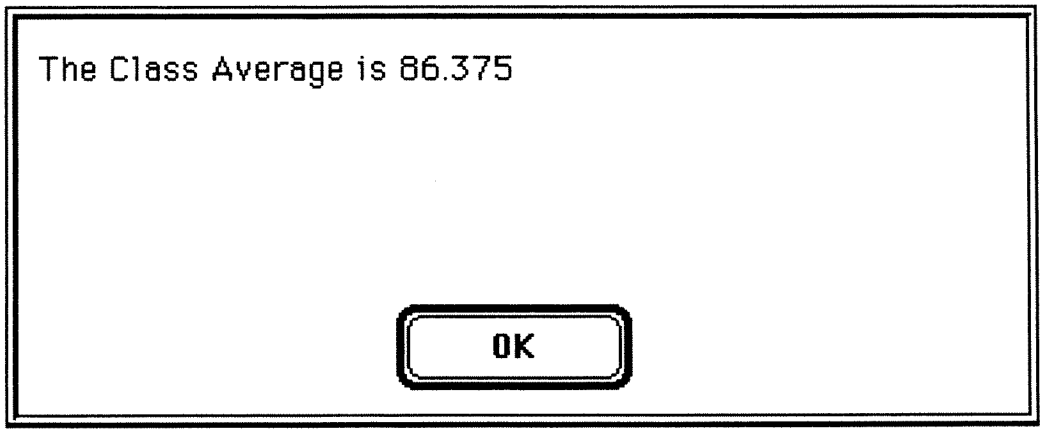

The Class Average is 86.375

Figure 55. Pop-up Dialog Box Displayed When Class Average Button Is Clicked.

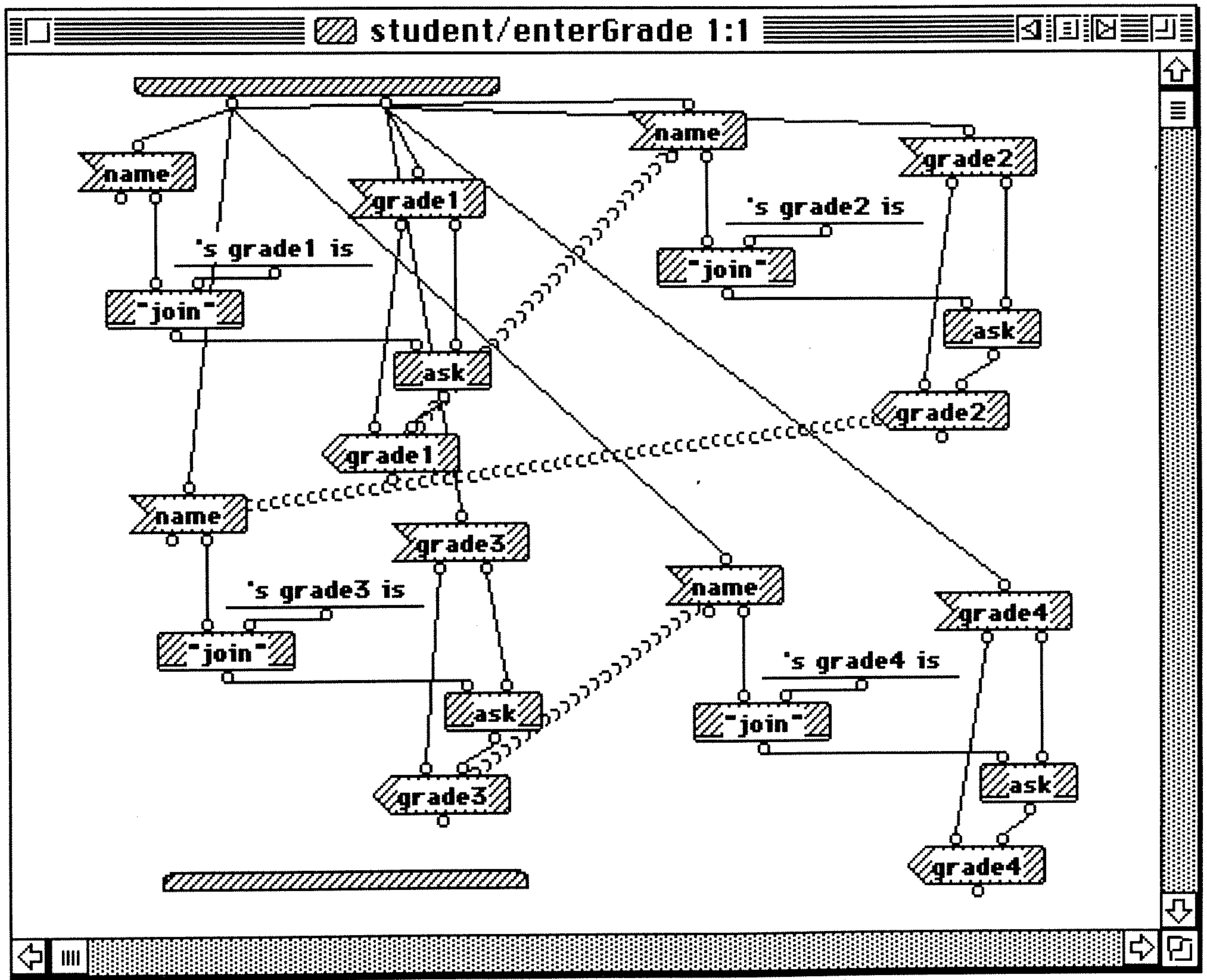


Figure 56. Method EnterGrade of Class Student.

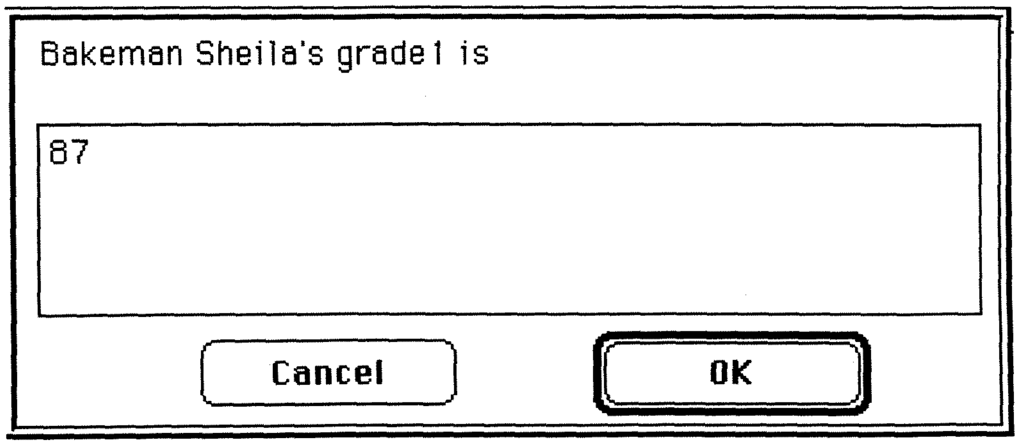


Figure 57. First of Four Pop-up Dialog Boxes Displayed When Enter Student's Grades Button Is Clicked.

The User window also includes a button labeled 'List Student's Grades,' located in the upper, right corner of Figure 53. This button, when clicked, will list the student's Grade1 through Grade4 attributes in the scroll block located below the button. The list of grades will also be displayed in the scroll box if the user double-clicks on the student's name. The method used to send the list of grades to the scroll box is Student/GradesToScroll (see Figure 58). In this method, the Grade1 through Grade4 attributes are converted from numbers into strings and packed into an array. The Window System Class attribute 'value list' is then set to this array. By setting the 'value list' attribute, the scroll list is updated to the list of student's grades.

Two additional buttons, 'Remove Student' and 'Student Average,' are included in the User window as is shown in Figure 53. 'Remove Student,' when clicked, deletes the highlighted student's name from the Gradebook persistent. 'Student Average,' when clicked, produces a dialog box with the student's name and average of grades. Figure 59 is an example of this dialog box.

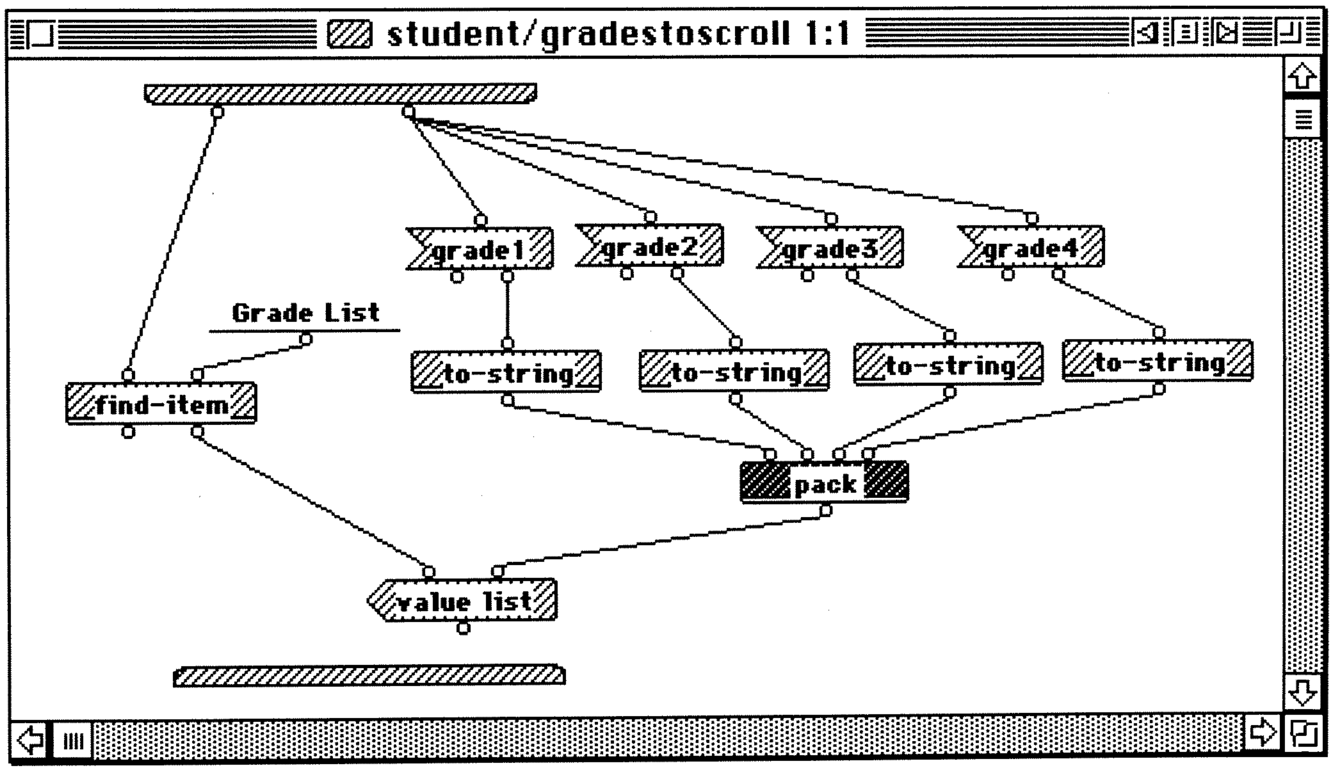
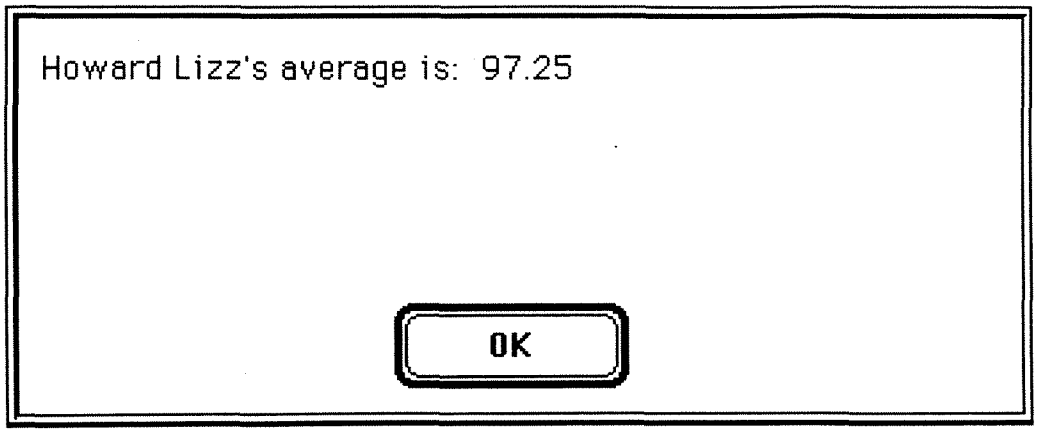


Figure 58. Method GradesToScroll of Class Student.



Howard Lizz's average is: 97.25

Figure 59. Pop-up Dialog Box Displayed When Student Average Button Is Clicked.

As has been illustrated, the development of a ProGraph Macintosh application is an involved process. The ProGraph environment does provide access to the necessary attributes to build an application, but the development time can still be lengthy.

7.0 Conclusion

7.1 Evaluative Criteria

The introduction of the visual programming paradigm necessitates a method of evaluation in order to provide a useful perspective. To provide a perspective, an extensive set of evaluative criteria has been developed and presented in Figure 12. As technological advances emerge, previously introduced visual languages will require reevaluation. When introduced, assembly code languages were heralded as significant advancements in language level when compared to machine code. As languages continued to grow in complexity and ability, assembly code languages were re-classified as low-level languages. Any method of evaluation of visual languages must therefore provide the flexibility to incorporate the inevitable technological leaps. The evaluative criteria and their ranges developed in this paper provides that flexibility.

The developed criteria were utilized to compare two commercially available visual languages. The evaluative criteria proved extremely useful when comparing the two languages. Since both languages provide many features, comparing them without a defined set of attributes would have been essentially impossible.

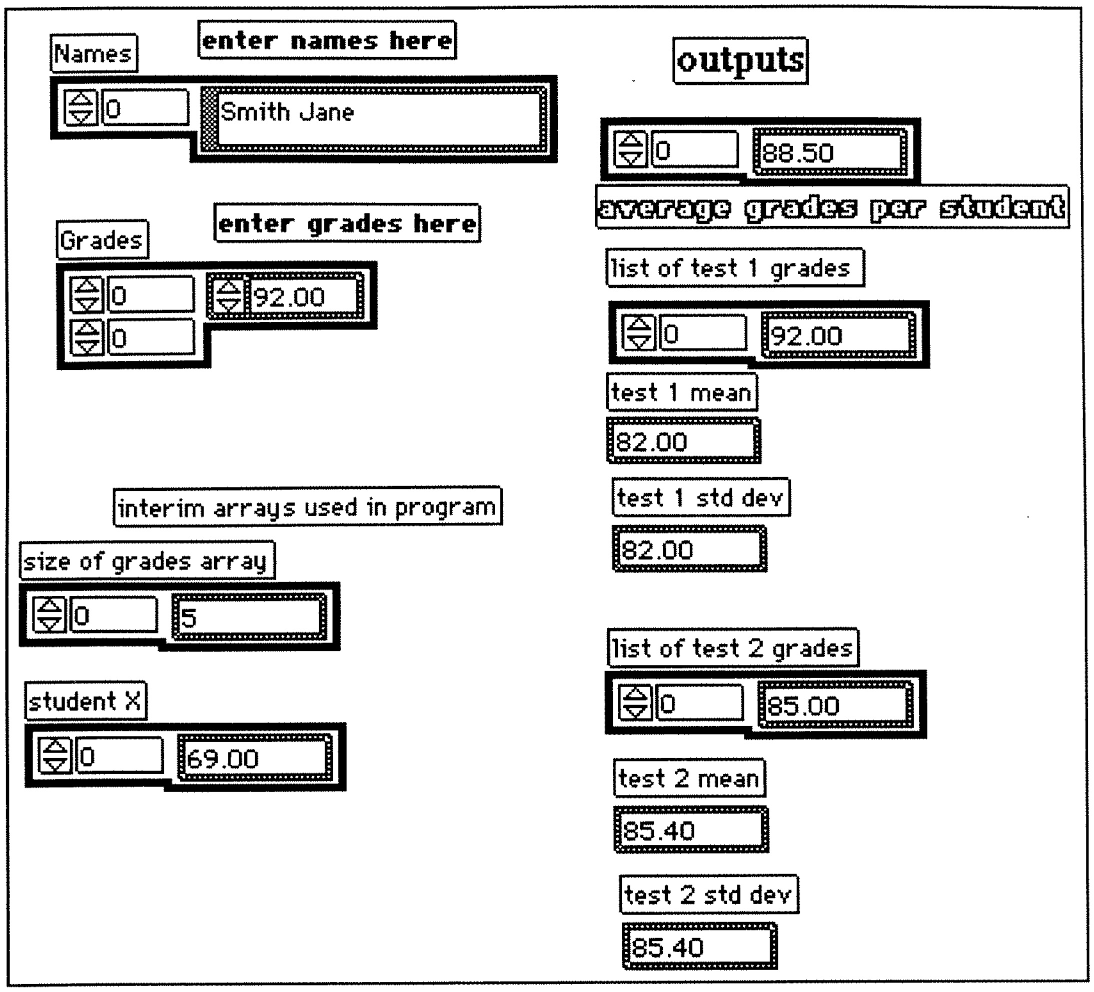
7.2 LabVIEW versus ProGraph

The concepts, capabilities, and example implementations of both LabVIEW and ProGraph have been presented. In some areas, such as in visual presentation, ease of use, and analysis performance, LabVIEW is clearly the preferred language. However, general applications in LabVIEW are very difficult to implement. For instance, a small portion of the Gradebook application was developed in LabVIEW. The front panel of the Gradebook Virtual Instrument and the corresponding diagram are presented in Figures 60 and 61, respectively.

The front panel, although functional, makes data entry very difficult. The user is not prompted for grades and is not prevented from entering grades in unacceptable positions in the array. All arrays used within the VI are visible on the front panel. These arrays could have been placed outside the normal viewing area of the screen, but scrolling the screen would have revealed the interim arrays. In comparison to the ProGraph version with pull-down menus and dialog boxes, the LabVIEW implementation is cumbersome at best.

The diagram of the Gradebook VI presented in Figure 61 reveals a significant deficiency in LabVIEW. The Name array and corresponding Grade array are not linked implicitly. An array cannot have elements of mixed type, therefore, separate arrays must be maintained. The inability to define complicated data structures would force the designer to maintain the link between Name and Grade explicitly. In this implementation, no link has been established. If the Name array were to be sorted into alphabetical order, the Grade array would no longer be valid. A sort VI would have to be developed to maintain the link as the arrays are sorted. This would be a difficult task, especially for the intended audience (test system design engineers, not computer scientists). In short, general applications in LabVIEW are complicated and difficult to implement.

Conversely, while ProGraph is the preferred language for general applications, there is no corresponding ProGraph application which can be practically designed to compare with the LabVIEW Spectrum Analyzer Virtual Instrument. ProGraph has no built-in analysis or data presentation features. All of these functions would have to be developed explicitly using ProGraph.



Names
enter names here
0
Smith Jane

Grades
enter grades here
0
0
92.00

interim arrays used in program

size of grades array
0
5

student X
0
69.00

outputs
0
88.50
average grades per student

list of test 1 grades
0
92.00

test 1 mean
82.00

test 1 std dev
82.00

list of test 2 grades
0
85.00

test 2 mean
85.40

test 2 std dev
85.40

Figure 60. LabVIEW Front Panel of Gradebook VI.

The most applicable areas for LabVIEW would include:

- data acquisition
- process control
- automated test systems
- presentation of visual programming concepts

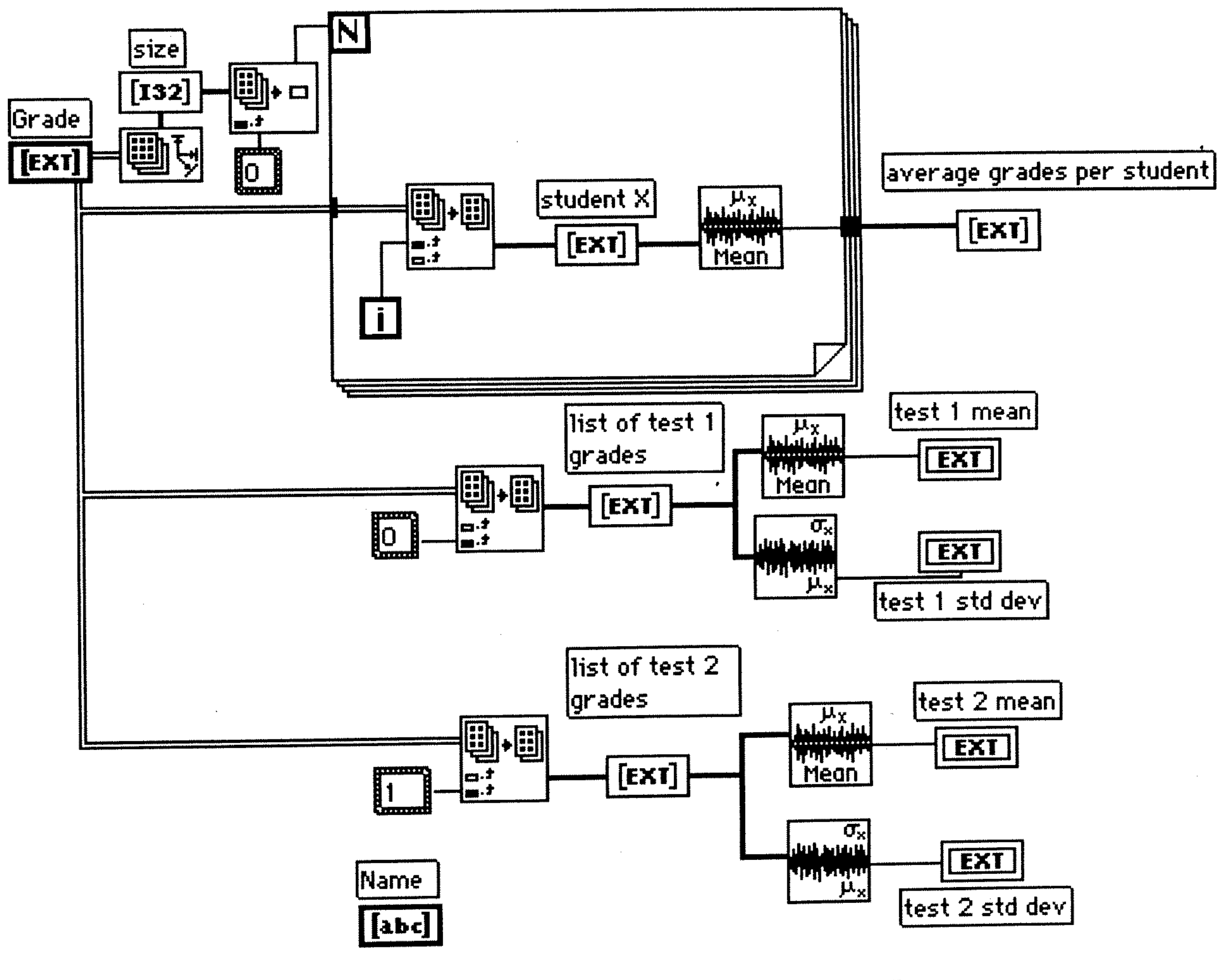


Figure 61. LabVIEW Diagram of Gradebook VI.

The most applicable areas for ProGraph would include:

- general programming applications
- object-oriented design
- editors
- presentation of object-oriented concepts

Both languages offer strong insight into the visual programming paradigm concepts and implementation issues. Useful applications can be developed in both LabVIEW and ProGraph. These are not toy languages, but languages which can produce complicated applications.

7.3 Usefulness of the Visual Programming Paradigm

In conclusion, a well-designed visual programming language reduces the complexity of the programming task and increase programmer productivity [Ames et al 1993; Faconti and Paterno 1992; Glinert and Tanimoto 1984; Myers 1986; Singh and Chignell 1992; Glinert 1990A; Glinert 1990B]. Visual programming enables the programmer to transfer concepts directly from his or her mind to the computer since the programming abstractions have been replaced by visual images. As Miller [1957] points out, the human mind is able to store 7 +/- 2 blocks of information concurrently, therefore, it is easy to conclude that by encapsulating several syntactic and semantic textual rules into one image, a programmer will be able to retain more information. This is especially true if that information coincides with the programmer's mental image. Furthermore, "images are easily learned, retained, and recalled as single units, often serving as the entire means of communication." [Glinert and Tanimoto 1984]. Therefore, visual programming may "provide a high bandwidth for human-machine communication." [Glinert and Tanimoto 1984].

Traditional textual languages have been based on heavily on the conventions of Indo-European languages, where abstract symbols are combined according to some linear syntax to form linear strings [Cox and Pietrzykowski 1988]. These abstract symbols are a severe restriction to people whose natural language is based upon graphical representations, such as Chinese. Visual programming offers an alternative which can span different cultures.

Additionally, visual programming is also well-suited to the object-oriented paradigm. Although visual programming is not inherently object-oriented [Winblad 1990], it is a logical step to provide object-oriented functionality to the user, as ProGraph clearly demonstrates.

The profile of the typical computer user has changed significantly in the last decade. Application development is no longer the sole domain of the computer scientist. Novice users are beginning to develop their own applications. There is a strong need to ease the process of software design. Visual programming may offer a solution to this problem. In addition, as computer graphics capabilities continue to improve, it is a logical step to take advantage of this advanced technology. Visual programming has the potential to be the partner to the growing hardware capabilities.

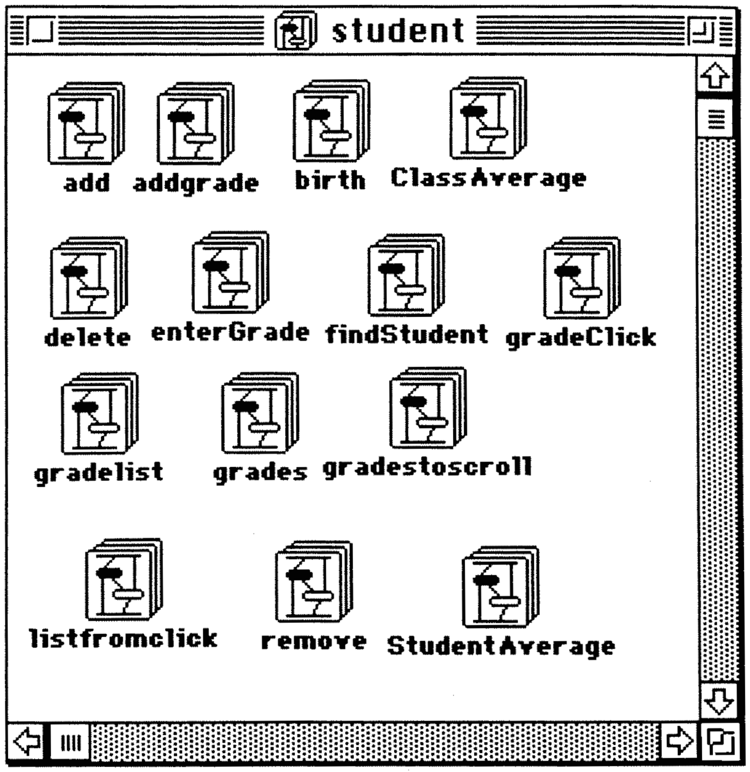
8.0 Reflections

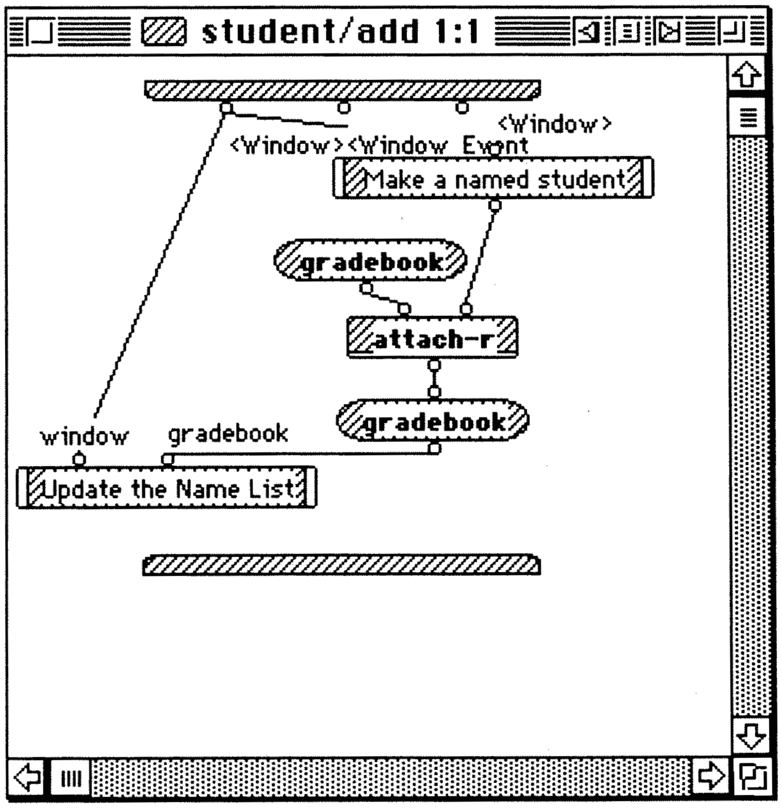
The focus of my thesis evolved during the last two semesters. Initially, my interest in visual programming lay with developing instrumentation applications using LabVIEW. Having designed numerous automated test and process control systems during my career, I was especially intrigued by a software package claiming to make that design process an easier endeavor. I had previously evaluated a sample of test system software introduced in the late 1980s and had found them lacking in either functionality or ease-of-use. I was pleased to discover that LabVIEW offered both strong functionality and was fairly easy to use.

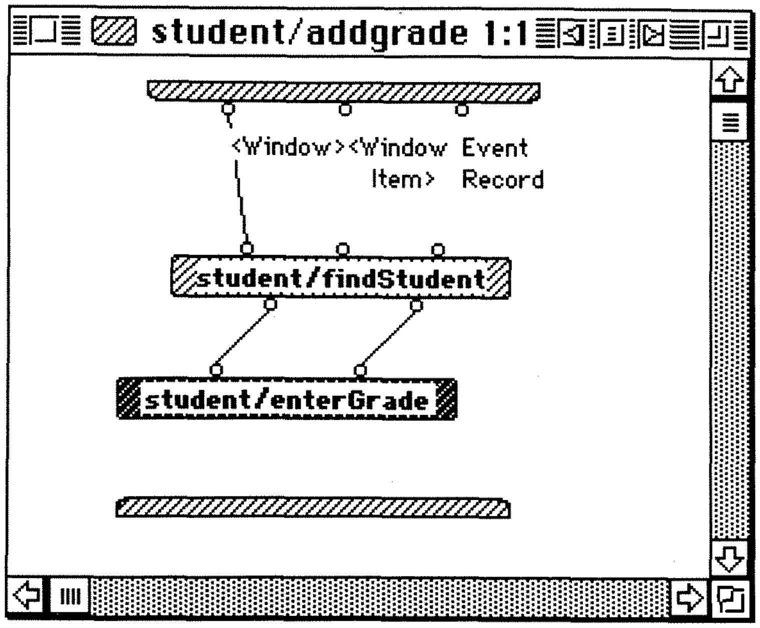
After evaluating LabVIEW and having read extensively about visual programming, I was anxious to determine how a more general programming language might be implemented. This lead me to ProGraph, not only because it was a visual language, but also because of the object-oriented paradigm. Working with ProGraph was more challenging for me. My understanding of object-orientation has increased significantly by evaluating ProGraph. Object-oriented concepts were easier to comprehend when presented visually.

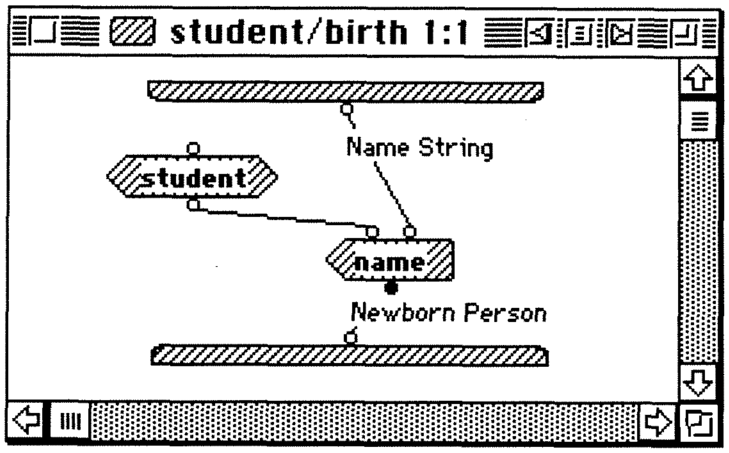
Upon reflection, I discover that I have learned a number of things. Among the most important is learning the object-oriented paradigm. Beyond that, I have learned how to systematically compare one software package with another. Also, I have learned how to organize my research, applications, and results in an understandable (hopefully) manner. Writing the thesis reinforced my belief that simply knowing a subject is not sufficient - I must be able to effectively communicate that knowledge. And, finally, I have learned that time-management is a crucial part of the thesis experience.

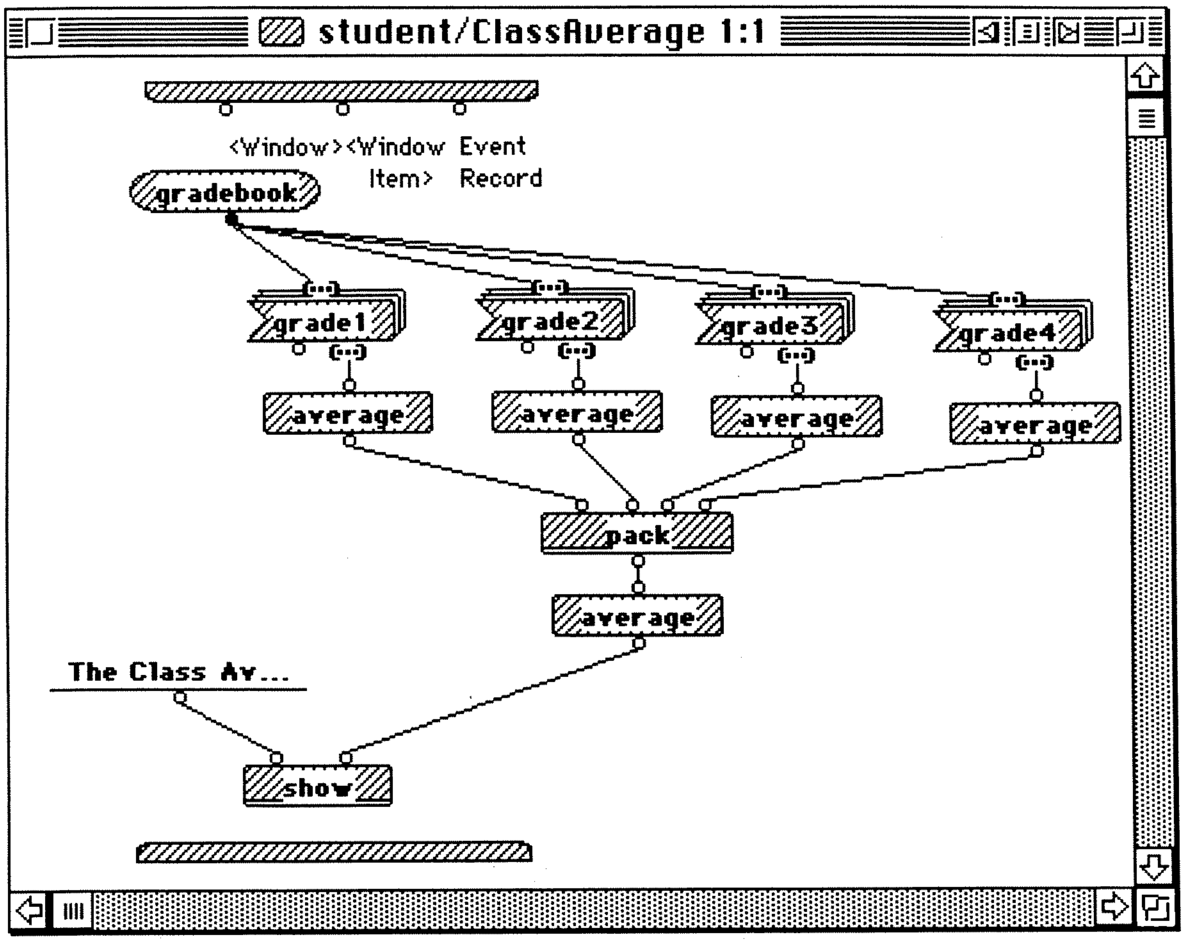
Appendix A. Student Class Method Diagrams of ProGraph Gradebook Application

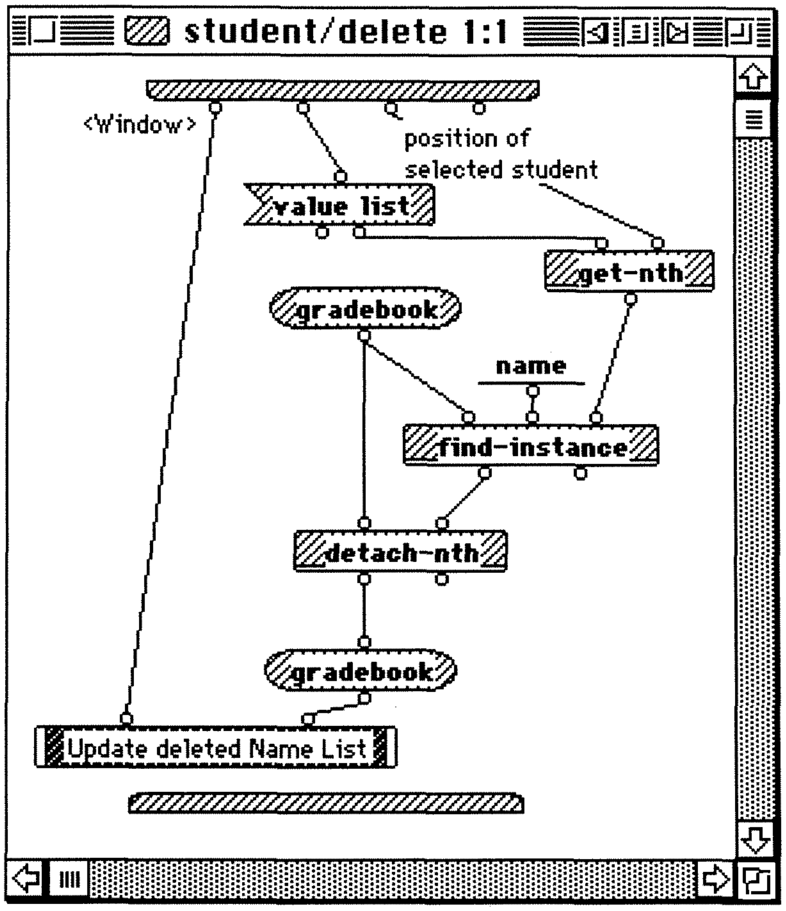


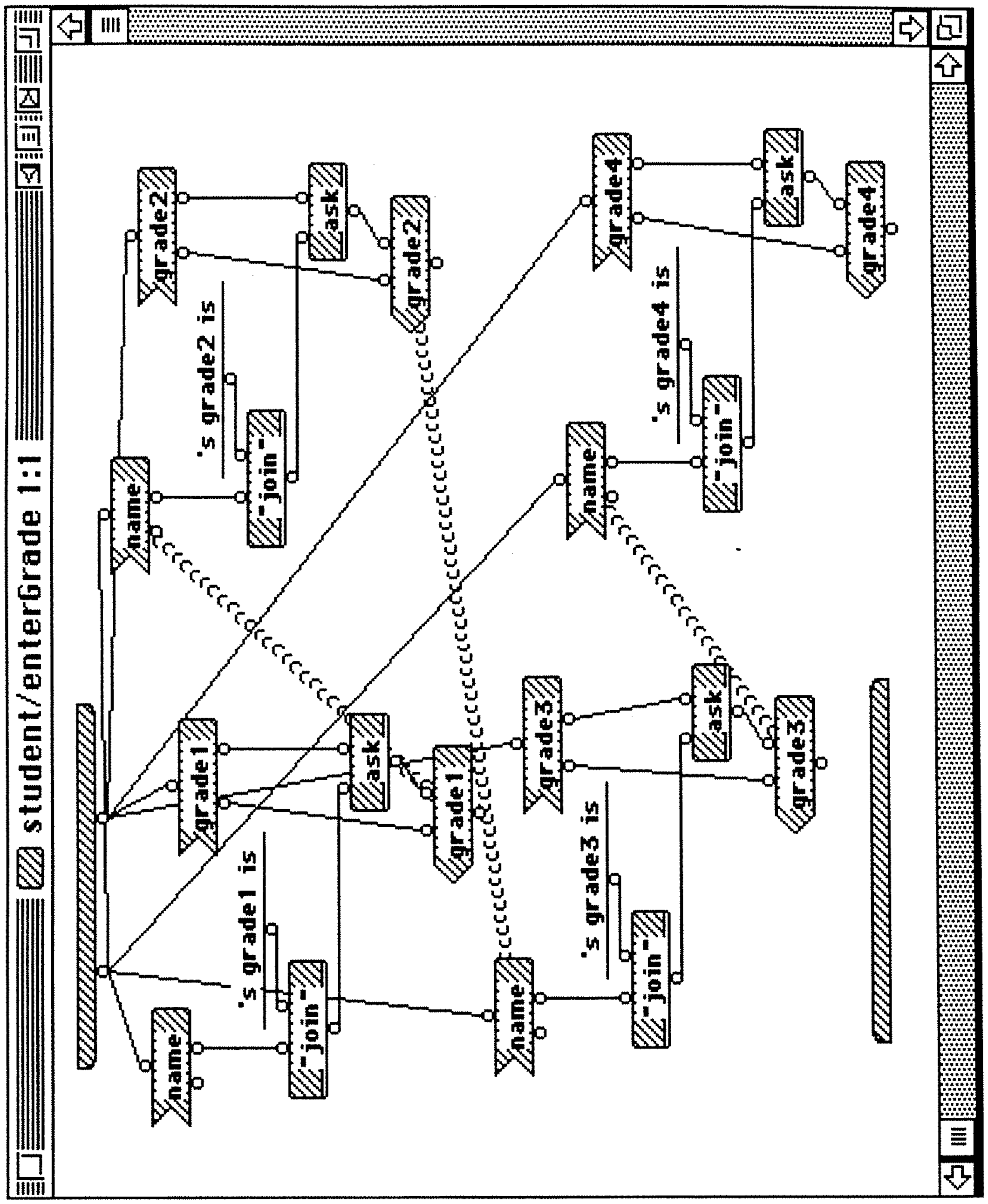


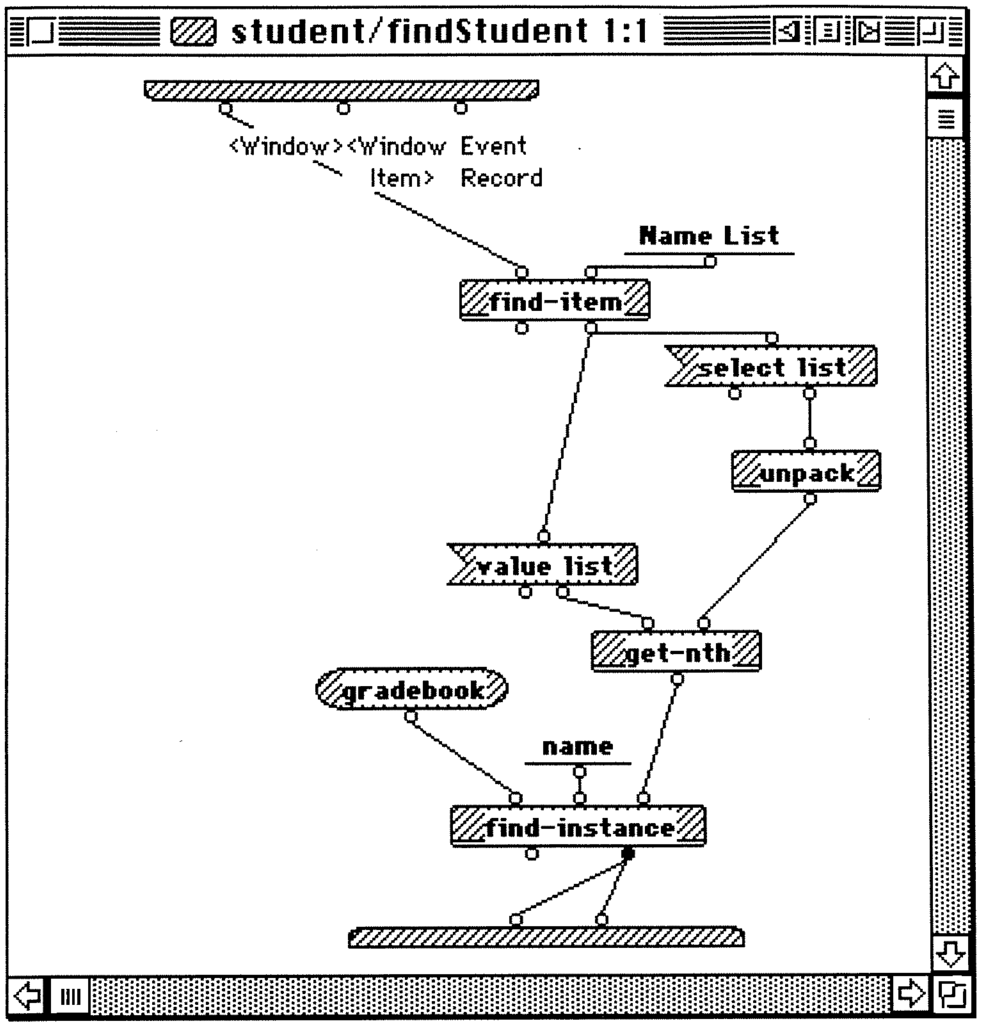

student/addgrade 1:1
<Window>
<Window Item>
Event Record
student/findStudent
student/enterGrade

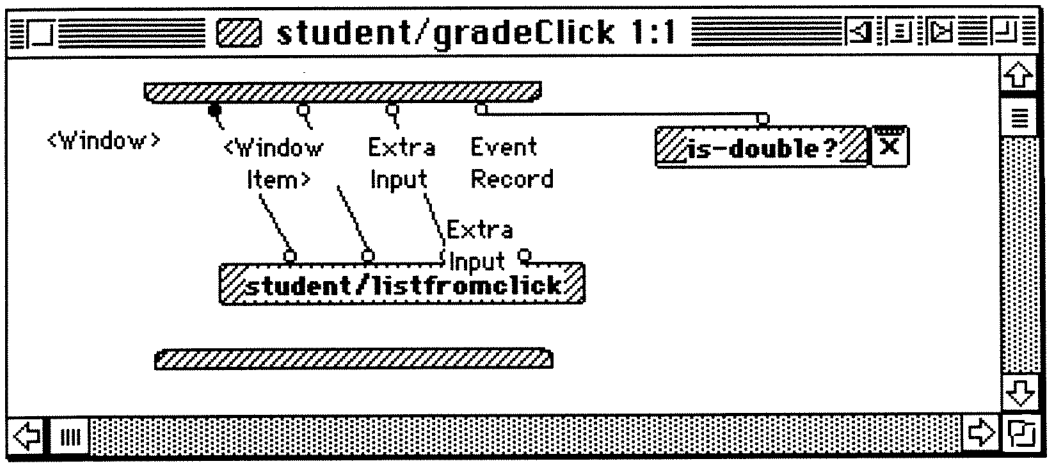

student/birth 1:1
Name String
student
name
Newborn Person

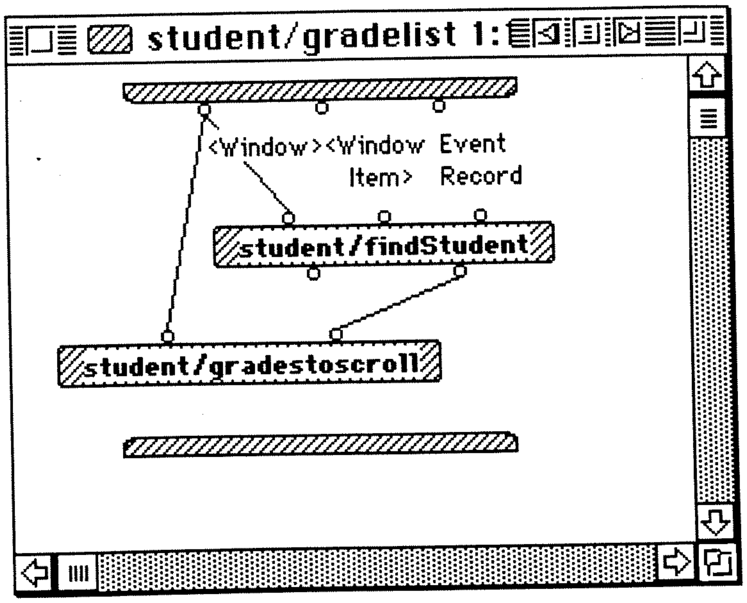

student/ClassAverage 1:1
<Window>
<Window Item>
Event Record
gradebook
grade1
grade2
grade3
grade4
average
average
average
average
pack
average
The Class Av...
show

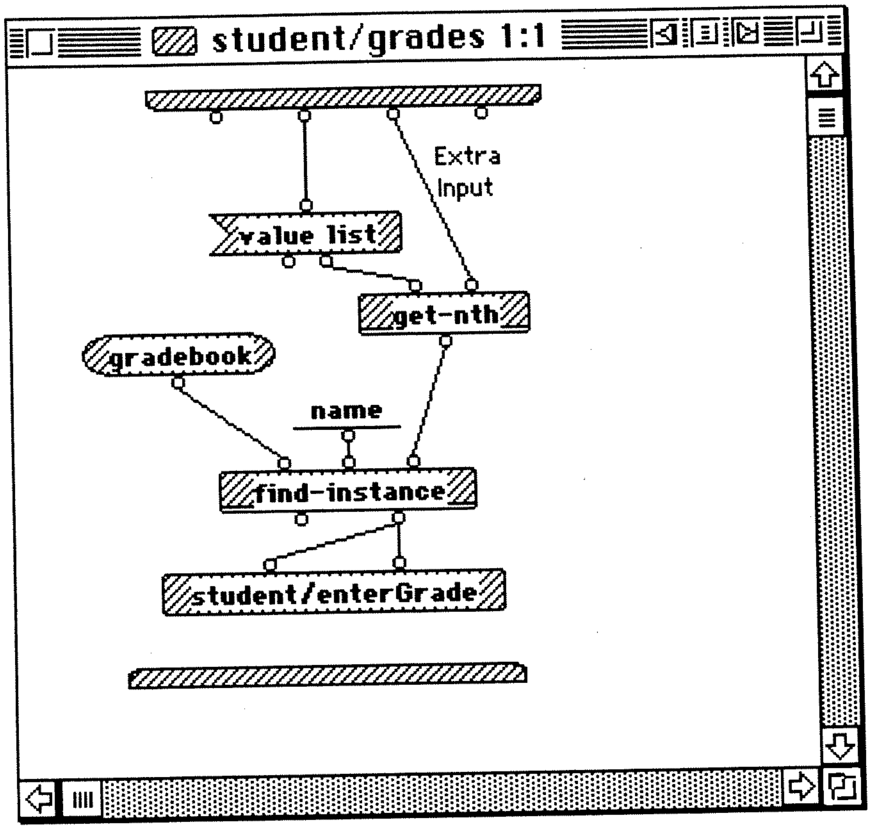

student/delete 1:1
<Window>
position of
selected student
value list
get-nth
gradebook
name
find-instance
detach-nth
gradebook
Update deleted Name List

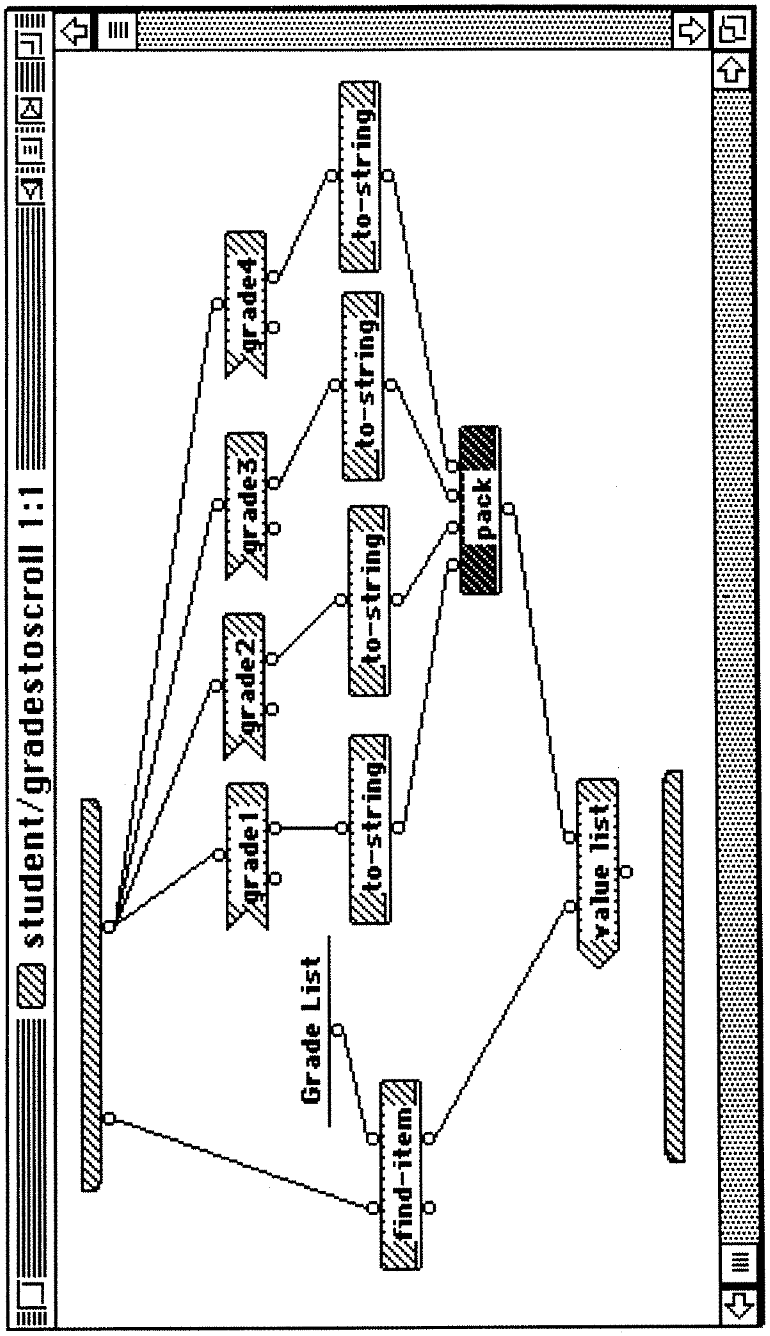

student/enterGrade 1:1
name
"join"
's grade1 is
grade1
ask
grade1
name
"join"
's grade3 is
grade3
ask
grade3
name
"join"
's grade2 is
grade2
ask
grade2
name
"join"
's grade4 is
grade4
ask
grade4

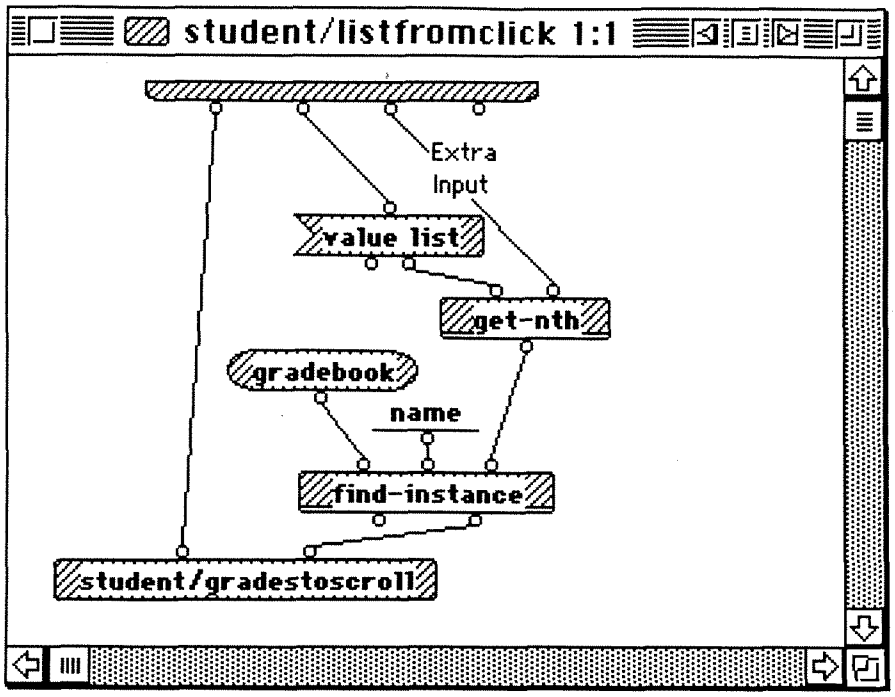

student/findStudent 1:1
<Window>
<Window Item>
Event Record
Name List
find-item
select list
unpack
value list
get-nth
gradebook
name
find-instance

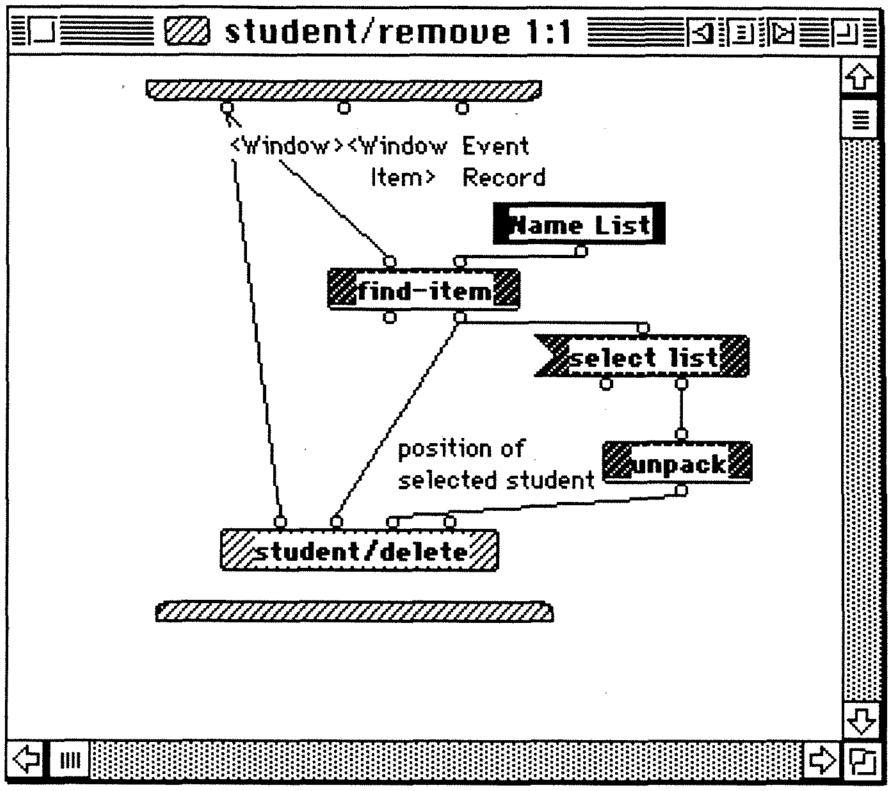

student/gradeClick 1:1
<Window>
<Window Item>
Extra Input
Event Record
is-double?
Extra Input
student/listfromclick

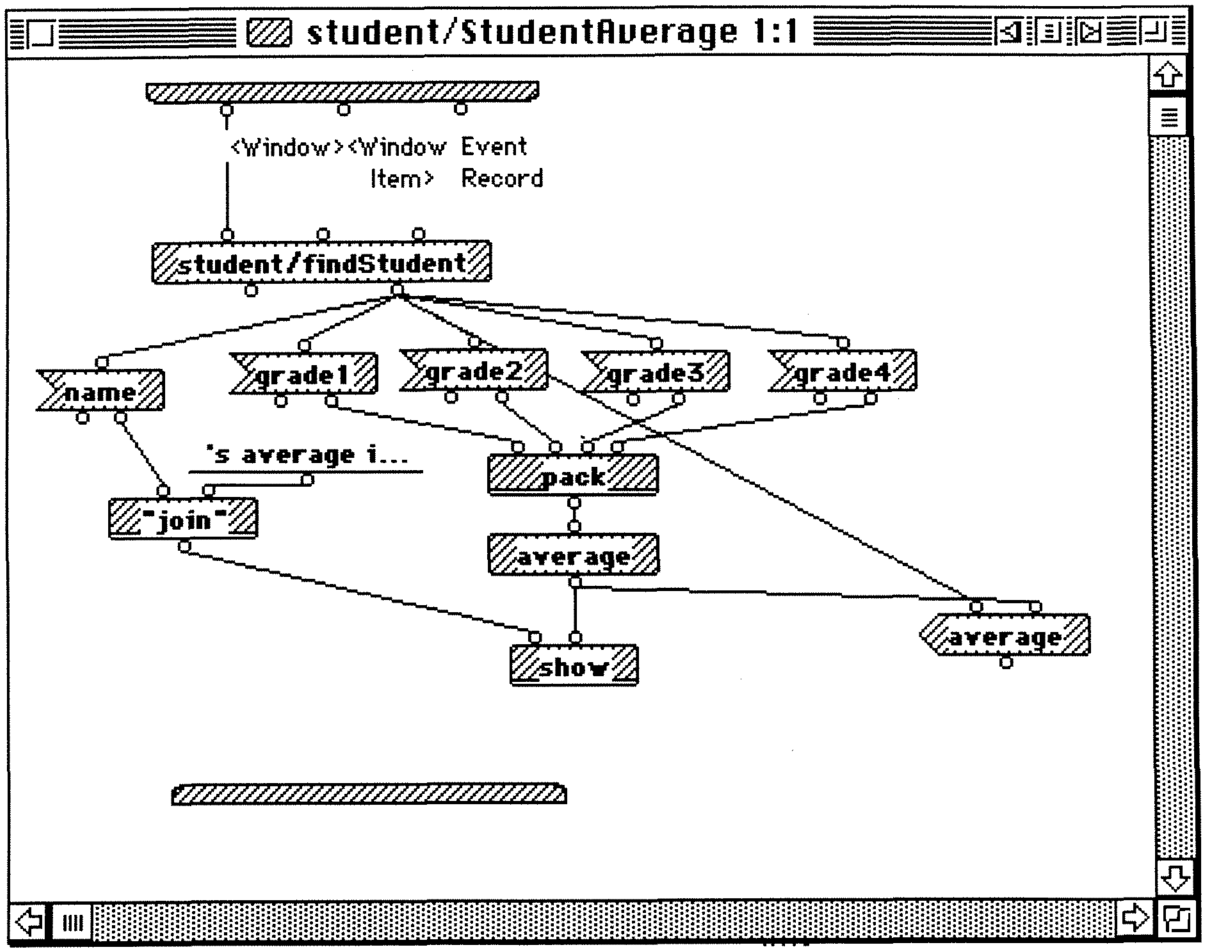

student/gradelist 1:
<Window>
<Window Item>
Event Record
student/findStudent
student/gradestoscroll


student/grades 1:1
Extra Input
value list
get-nth
gradebook
name
find-instance
student/enterGrade


student/gradestoscroll 1:1
grade1
grade2
grade3
grade4
to-string
to-string
to-string
to-string
pack
Grade List
find-item
value list


student/listfromclick 1:1
Extra
Input
value list
get-nth
gradebook
name
find-instance
student/gradestoscroll


student/remove 1:1
<Window>
<Window
Item>
Event
Record
Name List
find-item
select list
unpack
position of
selected student
student/delete


student/StudentAverage 1:1
<Window>
<Window Item>
Event Record
student/findStudent
name
grade1
grade2
grade3
grade4
's average i...
"join"
pack
average
show
average

Bibliography

[Ames *et al* 1993] Ames, Chuck, Kiper, James D., Auernheimer, Brent, and Burnett, Margaret. "V - A Visual Syntax for C," *Jet Propulsion Laboratory, Proposal to the Director's Discretionary Fund FY94*, August 1993.

This document proposes the development of a visual programming language V. V will have the equivalent visual syntax as the C programming language and a tool, C2V, would be developed to produce equivalent visual displays of existing C programs. The tentative benefits of V, as well as a suggested agenda for V's development, are presented.

[Baecker and Marcus 1990] Baecker, Ronald M. and Marcus, Aaron. *Human Factors and Typography for More Readable Programs.* Reading: Addison-Wesley Publishing Company, 1990.

This text introduces the concept of using tools to make a computer program easier to read. It covers the evolution of human factors in programming from *pretty printing* (using indentation and spacing to delineate different programming structures and embedding of structures) to graphical representations.

[Blackburn 1970] Blackburn, James A., ed. *Spectral Analysis: Methods and Techniques.* New York: Marcel Dekker, Inc., 1970.

This book presents the concepts of spectral analysis and the underlying mathematical principles. Detailed mathematical proofs are not presented, but the basic mathematical principles are well presented.

[Burnett and Baker 1993] Burnett, Margaret M. and Baker, Marla J., "A Classification System for Visual Programming Languages," *Technical Report 93-60-14*, Oregon State University, June 1993.

This paper presents a detailed classification scheme for classifying visual programming language research papers. This classification fills a void in the current ACM classification scheme for computing reviews. An overall hierarchy for visual computing is also included. This classification is a strong aid in clarifying the classification of systems labeled as 'visual programming.'

[Chang 1987] Chang, Shi-Kuo, "Visual Languages: A Tutorial and Survey," *IEEE Software*, Volume 4, Number 1, January 1987, pp. 29-39.

Chang introduces the concept of visual languages and subdivides them into four types: (a) languages that support visual interaction, (b) visual programming languages, (c) visual information processing languages, and (d) iconic visual information processing languages. Example languages are then introduced for the four categories.

[Chang 1990] Chang, Shi-Kuo, ed. *Visual Languages and Visual Programming.* New York: Plenum Press, 1990.

This book is divided into three parts: theory of visual languages, working examples of visual programming systems, and applications of visual languages and visual programming systems. Part I (Chapters 1 5) is devoted to the theory of visual languages. The five chapters cover iconic visual languages, diagramming languages, and formal semantics of a specialized visual language to specify operating system security. Part II (Chapters 6-12) covers several operational visual programming systems. The six chapters describe visual programming systems for parallel programming, program animation, construction of programmed learning software, experimental systems design, matrix-oriented programming, and open-ended graphical programming. The last five chapters in Part III (Chapters 12-16) present various applications of visual languages, visual programming, and visualization. In their design of a visual language for browsing, undoing, and redoing graphical interface commands, Kurlander and Feiner introduced the notion of an *editable graphical history.* An editable graphical history allows the user to review and modify the actions pre-formed with a graphical user interface. Using a pictorial metaphor borrowed from comic strips, an editable graphical history consists of a series of panels that depict in chronological order the important events in the history of a user's session. The user may scroll through the sequence of panels, reviewing actions at different levels of detail, and selectively undoing, modifying, and redoing previous actions. The graphical editor Chimera is described. By combining the notions of editable graphical history and dynamic icons, we can predict that such visual programming systems will be extremely useful for adaptable system simulation.

[Cox and Pietrzykowski 1988] Cox, P.T. and Pietrzykowski. "Using a Pictorial Representation to Combine Dataflow and Object-Orientation in a Language Independent Programming Mechanism." *Proceedings International Computer Science Conference*, 1988, pp. 695-704.

The standard textual representation of programming languages has many shortcomings, such as the abstract syntax inherited from Indo-European languages, enforced sequentiality, the necessity for variables, and the confusion between logical and mnemonic information. The AI languages Lisp and Prolog are improvements over the standard Algol-like languages, but still suffer from some of their drawbacks. The use of a pictorial representation for programming is proposed as a means for overcoming all of these shortcomings, incorporating the powerful features of AI languages and removing the bias towards Indo-European languages, making programming equally accessible to users whose natural language relies on ideograms, such as Chinese. The language ProGraph 2 is described using extensive examples, and the environment provided by the present implementation is briefly discussed.

[Cox *et al* 1989] Cox, P.T., Giles, F.R., Pietrzykowski, T. "ProGraph: A Step Towards Liberating Programming From Textual Conditioning." *1989 IEEE Workshop on Visual Languages*. Washington: IEEE Computer Society Press, 1989.

A critique of textual programming languages and software development environments, linking them to the development of hardware and discussing their connection with natural languages and mathematical formalisms. Criteria are outlined for modern integrated programming languages and environments based on the use of graphics. These principles are then illustrated by a description of the pictorial, dataflow, object-oriented language ProGraph and its implementation.

[Faconti and Paterno 1992] Faconti, G. P. and Paterno, F., "A Visual Environment to Define Composition of Interacting Graphical Objects," ***The Visual Computer*****, Volume 9, Number 2, September 1992, pp. 73-83.**

This work presents the FP visual language that specifies the components of a user interface and their relationship. Each component is an instance of an interactor that is a general description of a basic graphical interaction. By a visual language, it is possible to specify in a flexible way the logical structure of a user interface defined as a composition of interacting graphical objects. The graphical tool allows the designer to investigate the correctness of user interfaces and their properties.

[Fichman and Kemerer 1992] Fichman, Robert and Kemerer, Chris. "Object-Oriented and Conventional Analysis and Design Methodologies: Comparison and Critique." ***IEEE Computer*****, Volume 25, Number 10, October 1992, pp. 22-39.**

In this paper, several conventional analysis and design methodologies are briefly introduced and compared to each other. Object-oriented analysis and design methodologies are then introduced and compared to each other. Finally, conventional methodologies are compared to the object-oriented methodologies, illustrating where each paradigm has its strengths.

[Glinert 1990A] Glinert, Ephraim P., ed. ***Visual Programming Environments : Applications and Issues.*** **Los Alamitos: IEEE Computer Society Press, 1990.**

The second volume of the tutorial of visual programming environments focuses on implementations of various visual systems in chapters 1-4. Chapters 5-9 introduces the major issues of visual system design, such as: effective icon design, when graphics should be used over text, effects on the physically challenged, and the lack of formal definitions. Potential future applications are also presented.

[Glinert 1990B] Glinert, Ephraim P., ed. ***Visual Programming Environments : Paradigms and Systems.*** **Los Alamitos: IEEE Computer Society Press, 1990.**

The first volume of the tutorial of visual programming environments introduces the concepts and definitions of visual programming. Some of the traditional and non-traditional graphical representations are explored. Examples of iconic and visual extensions to mainline textual languages systems are presented as well as visual parallel and distributed computing environments.

[Glinert and Tanimoto 1984] Glinert, Ephraim P. and Tanimoto, Steven L. "Pict: An Interactive Graphical Programming Environment," ***IEEE Computer*****, Volume 17, Number 11, November 1984, pp. 7-25.**

In this paper, The authors propose that Pict, an interactive, graphically oriented programming environment is a more natural way for novices to learn programming than conventional, text-based programming languages. Pict's capabilities, design, and acceptance by novice users are presented.

[Miller 1957] Miller, G.A. "The Magic Number Seven Plus or Minus Two: Some Limits on Our Capacity for Information Processing." ***Psychological Review,*** **Volume 63, Number 2, February 1957, pp. 81-96.**

This paper reports the results of studies performed on short-term memory retention. It indicates that if information can be grouped into blocks, more information can be retained. Short-term memory is capable of handling 7 +/- 2 blocks of information.

[Myers 1986] Myers, B. A. "Visual Programming, Programming by Example and Program Visualization: A Taxonomy." In ***Conference Proceedings, CHI'86: Human Factors in Computing Systems,*** **Boston, Mass. New York: ACM Press, April 13-17, 1986, pages 59-66.**

This paper attempts to provide more meaning to the terms "Visual Programming," "Program Visualization," and "Programming by Example" by giving precise definitions, and then using these definitions to classify existing systems into a taxonomy. A number of common unsolved problems with most of these systems are also listed.

[National Instruments 1990] National Instruments Corporation. ***LabVIEW 2 Analysis VI Library Reference Manual.*** **Austin: National Instruments Corporation, 1990.**

This manual is divided into the following chapters:

- Chapter 1: Introduction to Analysis in LabVIEW containing an overview of the LabVIEW Analysis library of virtual instruments, a description of how it is organized, instructions for accessing the Analysis VIs and obtaining on-line help, and a description of LabVIEW Analysis error-reporting.
- Chapter 2: Numerical Analysis VIs describes the three groups of Numerical Analysis VIs - Array Operations, Complex Arithmetic, and Miscellaneous.
- Chapter 3: DSP VIs describes the two groups of Digital Signal Processing (DSP) VIs-Pattern Generation, and DSP.
- Chapter 4: Digital Filter VIs describes the two groups Digital Filter VIs-Smoothing Windows, and Infinite Impulse Response Filters.
- Chapter 5: Statistical Analysis VIs describes the three groups of Statistical Analysis VIs-Descriptive Statistics, Curve Fitting, and Linear Algebra.
- Index: alphabetically lists each VI and important concept described in this manual and its page number.

This manual does assume significant prior knowledge of the concepts. Very little detailed description of the function is given other than inputs required and outputs provided.

[Price *et al* 1993] Price, Blaine A.; Baecker, Ronald M.; Small, Ian S., "A Principled Taxonomy of Software Visualization," *Journal of Visual Languages*, Volume 4, Number 3, September 1993, pp. 211-266.

In the early 1980's researchers began building systems to visualize computer programs and algorithms using newly emerging graphical workstation technology. After more than a decade of advances in interface technology, a large variety of systems has been built and many different aspects of the visualization process have been investigated. As in any new branch of a science, a taxonomy is required so that researchers can use a common language to discuss the merits of existing systems, classify new ones (to see if they really are new), and identify gaps which suggest promising areas for further development. Several authors have suggested taxonomies for these visualization systems, but they have been ad hoc and have relied on only a handful of characteristics to describe a large and diverse area of work. Another major drawback of these taxonomies is their inability to accommodate expansion: there is no clear way to add new categories when the need arises.

This paper presents a detailed taxonomy of systems for the visualization of computer software. This taxonomy was derived from an established black-box model of software and is composed of a hierarchy with six broad categories at the top and over thirty leaf-level nodes at four hierarchical levels. Twelve systems are described in detail and the taxonomy is applied to them in order to illustrate its features. A research agenda for future work in the area is presented.

[Shu 1988] Shu, Nan C. *Visual Programming.* New York: Van Nostrand Reinhold Company, 1988.

Nan Shu has written this book as a tutorial and survey of visual programming. She begins by explaining why a new programming paradigm is needed and how visual programming might fill that need. She then categorizes different branches of visual programming. An individual chapter is devoted to each of the branches of her visual programming hierarchy. A framework for assessing visual languages (branch of her hierarchy) is introduced. This three-dimensional framework is often referenced in other documents on the subject. She then devotes a chapter as to how she envisions the future of visual programming.

[Singh and Chignell 1992] Singh, Gurminder and Chignell, Mark H.. "Components of the Visual Computer," *The Visual Computer*, Volume 9, Number 3, September 1992, pp. 115-142.

This paper reviews three major technologies that provide a platform for visual computing. These technologies reflect the needs of various people who use visual computers: programmers, end users, and scientists. A taxonomy of visual computing based on these three types of users is presented. We begin with a discussion of important developments in visual programming and follow with discussions of visual interfaces and visualization. We conclude with a summary of visual computing's current status and identify critical areas of research that should be emphasized in future work.

[Stroustrup 1988] Stroustrup, Bjarne. "What Is Object-Oriented Programming?" *IEEE Software,* Volume 5, Number 5, May 1988, pp. 10-20.

Stroustrup presents the fundamentals of object-oriented programming by first defining procedural programming followed by the definition of data abstraction. Stroustrup indicates that a programming language must provide class structures encapsulating data and methods and class inheritance to be considered object-oriented.

[Tektronix 1989] Tektronix, Inc. *Spectrum Analyzer Fundamentals.* Beaverton: Tektronix, Inc., 1989.

The underlying concepts of spectrum analysis and spectrum analyzers are presented. Following an introduction of spectrum analysis, examples of typical control settings for a spectrum analyzer are presented. Finally, potential applications for spectrum analyzers are detailed.

[TGS 1990A] The Gunakara Sun Systems. *ProGraph: Reference.* Halifax: The Gunakara Sun Systems, Limited. 1992.

The Reference manual consists of two parts. The reference chapters provide an organized source of information about the ProGraph language, editor, interpreter, compiler, Application Builder, System classes, and the Macintosh Toolbox interface. The appendices provide information on adding code written in the C language to a ProGraph application, as well as a thorough specification of the syntax and semantics of ProGraph.

[TGS 1990B] The Gunakara Sun Systems. *ProGraph: Tutorial.* Halifax: The Gunakara Sun Systems, Limited. 1992.

This manual is divided into two parts: Part 1: Preliminaries and Part 2: Tutorials. Part 1 shows how to set up the ProGraph programming environment, presents a grand tour of the ProGraph environment, explores the conceptual foundations of ProGraph, and surveys the ProGraph language. Part 2 is a progressive, example-based presentation of the features of both the ProGraph language and its development environment.

[Watson and Watson 1991] Watson, Collin J. and Watson, R. Douglas. "Computer Graphics Representation of A Statistical Model Used with Computer-Aided Diagnosis," *Computers and Biomedical Research*, October 1991, pp. 576-583.

A description of computer graphics of a multidimensional model that is used with computer-aided diagnosis or prognosis is presented. The model is discussed and computer graphics of the model are developed. The computer graphics are suitable as visual supplements for presenting the computer-aided diagnostic model to individuals who may be inexperienced in multivariate statistics.

[Winblad 1990] Winblad, Ann et al. *Object-Oriented Software.* Reading: Addison-Wesley Publishing Company, 1990.

This book covers all aspects of object-oriented design and engineering: from definition of relevant terms to examples of applications and their implementations.

[Witte 1993] Witte, Robert A. *Electronic Test Instruments: Theory and Applications,* Englewood Cliffs: Prentice Hall, 1993.

This book covers basic measurement theory of electronic signals. Instruments covered include DVMs, signal sources, oscilloscopes, frequency counters, spectrum analyzers, wavemeters, network analyzers, and logic analyzers.