OREGON STATE UNIVERSITY COMPUTER SCIENCE DEPARTMENT

# OSU: A High Speed Software Development Environment

Sherry Yang
Dr. Ted G. Lewis
Department of Computer Science
Oregon State University
Corvallis, OR 97331-3902

89-60-21

# OSU: A High Speed Software Development Environment

by

**Sherry Yang**

A research paper

Submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

**Master of Science**

Major Professor: Dr. Ted G. Lewis

Completed June 8, 1989

# ABSTRACT

Several problems with user interface design and implementation have been identified: (1) user interfaces are difficult and time-consuming to design and implement; (2) most user interface management systems (UIMS) are themselves difficult to use by a programmer; (3) UIMS's have not been integrated with other tools that support structured design, coding and maintenance, thus failing to maximize programmer productivity.

In the Oregon Speedcode Universe (O.S.U.) project, we had taken the following approaches: (1) direct manipulation programming technique is used to address the problems with user interface design and implementation; (2) integration of UIMS with CASE tools; and (3) high-level program generation from scripts, and reusable components.

This report surveys some of the existing UIMS's and describes O.S.U., a high-speed software development system. The main emphasis of this work is the design and implementation of Structure Chart Editor in O.S.U.. The Structure Chart Editor has three unique features: 1) combination of functional decomposition with object-oriented design, 2) alternate architectural views, e.g. call graph, uses graph, object graph, and graphical display of procedures, 3) merging the user interface specification with design and coding specifications.

Experimental results suggest that the techniques employed by OSU can be used to develop 50-90% of an application without explicit programming yielding 2-10 fold productivity improvements.

# Acknowledgments

I would like to express my appreciation to Dr. Ted G. Lewis, for his guidance, understanding, and encouragement. Full credit for the origination of the idea for Oregon Speedcode Universe must go to Dr. Lewis. Working with him is a rewarding experience and I am looking forward to continue working with him.

I thank Dr. Walter Rudd and Dr. Bruce D'Ambrosio for serving on my committee and for their helpful comments.

I thank other members of the Oregon Speedcode Universe: Fred Handloser, Sharada Bose, Jagannath Raghu, Jihwan Lin, Ching-pin Liao, Kritawan Kruatrachue, James Armstrong, Molly Joy, John Chia and very special thank you to Chia-Chi Hsieh, who is always there when I needed a friend, no matter how far away she is.

I thank the "gang": Kirt Winter, Karl Schricker, Bob Singh, Becky Roof, and Paula Hannan for the good times we had.

Lastly, I would like to express my gratitude to my parents, grandma and my brother Samson for their support and encouragement.

# Table of Contents

# 0. Introduction

## A. The Problem

User interfaces are difficult and time-consuming to implement. Because they constitute approximately 70% of a typical application, they represent a major obstacle to software development. The easy-to-use direct-manipulation interfaces popular on many modern systems are among the most difficult to implement. These interfaces let the user operate directly on objects that are visible on the screen, performing rapid, reversible, incremental actions. Direct manipulation interfaces, such as those found on the Apple Macintosh, are difficult to create because they often provide elaborate graphics, many ways to give the same command, many asynchronous input devices, a mode-free interface (the user can give any command at virtually any time) and rapid *semantic feedback*. Semantic feedback is the appropriate response to user actions based on specialized information about the objects in the program.

Since user interfaces constitute a significant portion of a typical application, automating the production of user interface code seems to be the solution to the problem. Many user interface management tools and systems have attempted to do exactly that, but with little success. The main problems with the existing systems are they are too difficult to use, in that they usually require the programmer to know hundreds of procedures in a toolkit and/or special-purpose language or diagram to specify the user interface; and they offer too little functionality [Myers 89]. Most user interface management systems (UIMS) provide only a small part of the design task. While they are very good at handling menus and scroll bars, they can rarely be used to help control

the display and manipulation of the application's data objects. Many of these systems make no attempt to handle an application's output. In addition, most systems require the use of a special script language to provide functionality. This causes problems, because in additional to the learning curve, a programmer must revert to using low-level coding tools to do the complete application. Finally, UIMS's have not previously been integrated with other tools. For example, most Computer-aided Software Engineering (CASE) tools ignore user interface management systems, and separate design into the functional part and the interface part. In order to achieve the maximum increase of programmer productivity possible from these tools, UIMS must be integrated with other tools.

## B. The Approach

We suggest that the problems with user interfaces and current UIMS's can be overcome with a user interface management system based on direct manipulation programming technique, integration of UIMS with CASE, and automatic code generation.

Direct manipulation is a new programming concept. Its main principle is showing instead of telling. Telling is done by manuals, programming languages, and other written documents which attempt to teach user and machine alike. Showing is done by doing in the form of direct manipulation of "objects". No manuals, programs, or other written documents are needed and there is no linguistic ambiguity in showing because showing is direct. Showing a computer what to do is difficult, and at present, less successful than traditional methods of giving instructions by telling via a programming language. However,

showing has the potential for major advances in programmer productivity while programming by telling has reached a 20-year plateau [Musa 85]. Even an imperfect software tool for programming by showing can have dramatic impact on programmer productivity. In addition, direct manipulation programming is concrete, not abstract. It allows the user to work with objects of interest, user interface objects, directly, not through some abstraction notation or languages. Direct manipulation programming ideally provides a means for the programmer to use only the mouse to indicate what the programmer desires.

In order to incorporate functionality to the user interface of an application, CASE tools need to be integrate with user interface management systems. The CASE tools need to serve not only as automated design tools, but also automated program generator tools that automatically generate source code from design specifications. The CASE approach is chosen because of the many benefits that can be derived from it. It enables the reuse of software components, it speeds up development process, and it simplifies program maintenance. In sum, CASE improves programmer productivity.

Oregon Speedcode Universe (O.S.U.) is a high-speed software development environment for the Macintosh. Its has a UIMS that allows the user to prototype the user interface portion of an application very rapidly and automatically without explicit coding. O.S.U. is based on direct manipulation programming. By allowing direct manipulation of the user interface objects, O.S.U. frees the programmer from the need to learn new language or diagrammatic technique for specifying the

user interface. In contrast, the programmer must understand many of the 600+ ROM-based **toolbox** routines to write Macintosh applications.

The pair of CASE tools integrated with the UIMS portion of O.S.U. are : 1) Structure Chart Editor, which is a modular design tool with which the programmer specifies the modular structure of the application, and 2) the VIGRAM(VIsual proGRAMming) tool, which is a detailed design tool with which the programmer specifies the down-to-the-statement level detail of a procedure/function.

O.S.U., though limited in functionality in its current state, is designed for wide-spectrum prototyping. To illustrate its wide-spectrum prototyping functionality, current work is underway to design and implement several domain-specific software accelerators. [Lewis 89]

## C. Scope of Work

The design and implementation of the Structure Chart Editor, a programming environment which combines 2 structured design methods with user interface design, is the main scope of this work. Also included in this work is a survey of existing user interface management systems and comparison between earlier versions and the existing O.S.U. system.

## D. Significance of Work

Some preliminary results using O.S.U. indicate that direct manipulation programming, integration of UIMS and CASE and automatic program generation, O.S.U. can achieve a 3.3-fold increase in programmer productivity in limited use [Lewis 89]. Further

experimentation is needed before it is clear how powerful this approach is. However, we anticipate 2 to 10 fold improvments.

The CASE portion of the work is particularly significant in that it is the first CASE tool to combine structure charts from functional decomposition method with object-oriented design methodology. It is also the first CASE tool to combine user interface design with structured design methodologies.

# I. User Interface Management Systems (UIMS)

## A. Introduction

Creating good user interfaces for software is very difficult. Interface software is often large, complex, and difficult to debug and modify [Myers 89]. The user interface portion of software systems seems to be the bottleneck of software development. Consequently, many tools and systems have been developed in an attempt to ease their design and creation.

A user-interface tools come in two general forms: User interface toolkits and user interface management systems.

User interface toolkit is a library of interaction techniques, where an interaction technique is a way of using a physical input device (such as mouse, keyboard, tablet, or rotary knob) to input a value (such as command, number, percent, location, or name) along with the feedback that appears on the screen. Examples of interaction techniques are menus, graphical scroll bars, and on-screen buttons operated with the mouse. A programmer uses a user-interface toolkit by writing code to invoke and organize the interaction technique. Toolkits do not provide much automatic support for the design of interfaces or for the specification of sequencing and dialogue control.

A user interface management system (UIMS) is an integrated set of tools that help a programmer create and manage many aspects of a user interface. A UIMS helps with both designing and implementing the interface and so encompasses a broader class of programs than does a toolkit.

## 1. User interface toolkits

There are two kinds of toolkits: 1) a collection of procedures that can be called by application programs, like the Macintosh **toolbox**; and 2) a collection of objects with inheritance, which make it easier for the designer to customize interfaces, like the X.11 toolkit for the X window system manager. With all toolkits, the designer writes programs in a conventional programming language to control the interface. A toolkit typically includes hundreds of procedures that implement many interaction techniques.

The problem with using toolkits is that they provide limited interaction styles and are often expensive to create and difficult to use. It is often not clear how to use the procedures to create a desired interface. Hence, not only do we need a collection of procedures or objects, but we also need an environment that will aid in the design, implementation and maintenance of the user interface component of an application.

## 2. User Interface Management System (UIMS)

A UIMS, according to Hix [Hix 89], is a set of interactive tools for the development and execution of the user interface of a software system. In particular, they aid in specification, design, prototyping, implementation, execution, evaluation, modification, and maintenance of the user interface component of an interactive software system.

User Interface Management Systems [Cardwell 87] have 3 characteristics:

- They comprise a set of shared and reusable code modules, which are separate and independent from the application specific software.
- The shared or reusable code modules are capable of implementing an abstract or generalized set of user interaction or dialogue techniques. For example, the modules can provide command-line parsers, menuing systems, and forms systems. Moreover, each of these dialogue techniques can be implemented in a wide variety of applications.
- The UIMS code modules are general enough to work with a set of methods, techniques or tools for the description or specification of the user interface for a wide range of specific applications.

Hence a UIMS is not only a modular, application-independent user interface, but it is also a set of user interface code modules that may be reconfigured and reused with different application together with a collection of software tools that enable this to happen.

A UIMS helps the designer combine and sequence interactions between user and application. Some UIMS's help the designer create toolkits, while others help the designer lay out and use predefined user interface objects.

A comprehensive UIMS handles all aspects of the interface, which includes all visible parts of the display and all aspects of the dialogue between the user and the application. The UIMS should

- Handle the mouse and other input devices,
- Validate user inputs,
- Handle user errors,
- Process user-specified aborts and undos,

- Provide appropriate feedback to show that input has been received,
- Provide help and prompts,
- Update the display when application data changes,
- Notify the application when the user updates application data,
- Handle field scrolling and editing,
- Insulate the application from screen-management functions,
- Automatically evaluate the interface and propose improvements, or at least provide information to help the designer evaluate the interface, and
- Let the programmer customize the interface.

In order to perform these functions, the UIMS may contain

- A toolkit,
- A dialog-control component that handles event sequencing and interaction technique,
- A programming framework that helps guide and structure the interface code and application semantics,
- A mouse-based layout editor to specify the location of graphical elements, and
- An analysis component that may either evaluate the interface automatically based on rules and guidelines or save information such as keystrokes for later evaluation by the designer.

Using interface tools have 2 advantages in the following areas:

1. It results in better interfaces:

- Designs can be rapidly prototyped and implemented, possibly before the application code is written.

- It is easier to incorporate changes discovered through user testing because the interface is easier to modify.
- One application can have many interfaces.
- More effort can be expended on the user interface tools than may be practical on any single interface because the tools will be used again and again.
- Different applications will have more consistent interfaces because they have been created with the same user-interface tools.
- It is easier to investigate different styles for an interface, thereby providing a unique look and feel for a program.
- It is easier for many specialists to be involved in designing the interface, including graphic artists, cognitive psychologists, and human factors specialists. Professional interface designers, who may not be programmers, may be in charge of the overall design.

2. The interface code will be easier to create and more economical to maintain:

- The code will be better structured and more modular because it has been separated from the application. This lets the designer change the interface without affecting the application, it lets the programmer change the application without affecting the interface.
- The code will be more reusable because the user interface tools incorporate common parts.
- The reliability of the interface is higher because the code is created automatically from a higher level specification.
- Interface specifications can be represented, validated, and evaluated more easily.

- Device dependencies are isolated in the user-interface tool, so it is easier to port an application to different environments.

## B. Survey of UIMS

### 1. History of Development

The term UIMS was first used by Kasik in 1982 [Kasik 82]. However, the concept of tools for supporting development and execution of the user interface existed well before this time. In 1982, a workshop on Graphical Input and Interaction Techniques was held in Seattle, Washington [GIIT 83]. The purpose of this meeting was to understand and document an emerging change in technical emphasis on interactive graphics and interaction techniques in the human-computer interface. Results of this workshop included exposition of what constituted a UIMS, its role in the software development process, and an acknowledgement of the need for interdisciplinary research and exchange in this field. Emphasis at this workshop was heavily on run-time support for the interface with much less concern for design-time tools.

Closely following the Seattle workshop, another working meeting was held in Seeheim West Germany in 1983 [Pfaff 85]. Focus at this session included concentration on the role, model, structure and construction of a UIMS. A significant result of this workshop was what has become known as the "Seeheim model" that describes the logical components of a UIMS.

ACM SIGGRAPH sponsored a workshop, again in Seattle, Washington, in 1986. Its goal was to synthesize new ideas and directions for future research in software tools for interface

management. Working groups addressed topics such as goals and objectives of UIMS the relationship between the UIMS and the application, and the environment for UIMS. A main topic addressed was the process of designing human-computer interface, in particular, the methodological and software engineering issues, and the type of UIMS needed by interface developers to support these issues. This was apparently the first discussion of the role of a UIMS in the software development process and the relationship of a UIMS to software engineering. In particular, conclusions of the session included the kinds of tools and other support that were thought to be appropriate at all phases of the traditional waterfall life cycle of software development. Based on the success of the 1986 meeting, SIGGRAPH is now sponsoring symposia on UI Software. This area is receiving wide attention at various different conferences and journals.

## 2. Generations of UIMS

### a. First Generation.

First generation UIMS's were not really UIMS's, in any broad sense, but were more pre-UIMS-- the predecessors to modern UIMS's [HIX 89]. They were generally facade-only and simulation-like prototype builders that were capable of producing only a mock-up of an user interface. Once that mock-up was deemed satisfactory (by its developers usually, rather than by its intended end-users) it was thrown away and work was begun on developing the real software system, complete with a user interface, that was hopefully patterned after that of the mock-up. This generation also included display managers that provide general development tools for some parts of the

user interface, with emphasis on screen display. While they address some problems of interface design, display managers lack a generality in approach as they are typically oriented toward development of a specific format or interaction technique (e.g. menus or forms). They are also often limited to specific devices and specific classes of applications.

UIMS's of the first generation established themselves in the research arena as well as the commercial world. Interfaces produced by these UIMS were typically specified in a BNF-style language, supplemented with conventional programming. They were tools for application programmers, not for non-programming interface developers.

The first generation "pre-UIMS" were relevant to the world of software engineering in the sense that they were used by software developers to help simulate and build parts of what eventually became an application system. The ability to prototype an interface, even if it was only a facade, and to produce some parts (displays) of the interface without writing source code, demonstrated that such capabilities were a good conceptual foundation upon which to build and extend into a second generation of UIMS's.

b. Second Generation.

Second generation UIMS's focused on support for execution of the user interface, with very little emphasis on interface design, human factors, or the end-user of the systems these UIMS's produced [HIX 89]. The second generation produced several experimental systems that greatly expanded the conceptual knowledge and experience base for developing such systems (See below for some example systems). This

included departure from the BNF-like grammars used in the first generation UIMS into use of state-transition diagrams, declarative languages, event languages and graphical techniques for representation of the interface, formal abstraction of the interface from the rest of the application system, and increased varieties of interaction styles.

State-transition diagrams can be used to code the interface, because much of what an interface does involves handling a sequence of input events. A state-transition network is a diagram that represents the behavior of finite-state machines. A finite-state machine is a hypothetical mechanism which can be in one of a discrete number of conditions or states. Certain events can cause the finite-state machine to change its state. Events can occur asynchronously, that is, at any point in time, or synchronously, at clock intervals. An example of a state-transition diagram shown in Figure 1 is the representation of a simple desk calculator from **Jacob**[Jacob 85 ]'s State-Diagram Interpreter. The circles represent the states, and arcs out of each state are labeled with an input token, which is the event that will trigger the state transition. In addition to input tokens, the arcs in some systems are labeled with application procedures to be called and output to be displayed.

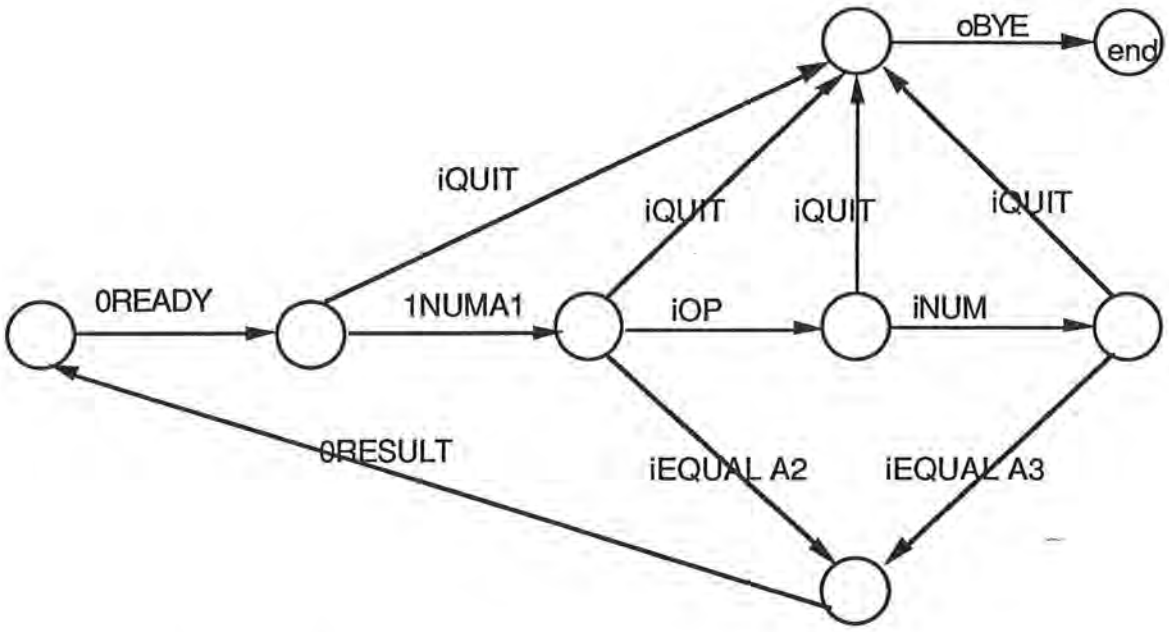



**Figure 1. A state-transition diagram from Jacob's State-Diagram Interpreter of a simple desk calculator. [Jacob 85]**

The problems with the state-transition approach are that the connections between the interface and application are made through global variables, and all states must have explicit arcs for all possible erroneous input and all universal commands such as Help and Undo.

State-transition UIMS's are most useful when the interface must do a lot of syntactic parsing or has many modes (each state is really a mode). However, most highly interactive systems are largely mode-free, so the user has many choices at every point. Because this requires many arcs out of each state, the state-transition method has not been successful in mode-less interface design. Another problem with state-transition networks is that they cannot handle interfaces that let the user operate on multiple objects concurrently (possibly using multiple input devices). Also the diagrams get very confusing when used for

large interfaces, because they become a maze, with arcs that are hard to follow as they go off the page or screen.

With an event language, input tokens are considered to be events that are sent immediate by event handlers. These handlers can cause output events, change the internal state of the system (which might enable other event handlers) and call application routines.

**Algae**[Flecchia 87] uses an event language that is an extension of Pascal. The designer programs the interface as a set of small event handlers, which **Algae** compiles into conventional code.

**Sassafras**[Hill 86] uses a similar idea but with an entirely different syntax. It uses local variables called flags to help specify control flow. **Sassafras** is especially well-suited for interfaces that use multiple input devices concurrently (also called multi-threaded dialogues). It can also support direct-manipulation interfaces because it promotes efficient, frequent communication between the UIMS objects and the application program.

**Squeak**[Cardelli 85] is a textual language for mouse-based interfaces that exploit concurrency. Squeak's processes are similar to event handlers and the messages sent by Squeak processes are similar to events. Squeak supports many concurrently active input devices. The primitive input events are mouse-button transitions, keyboard key processes, incremental movements of the mouse or other devices, and clock time-outs.

A Squeak program compiles into a sequential state machine. Although it is a compact notation for specifying complex, time-dependent interfaces, Squeak is unfortunately a fairly difficult language to write in.

Event-language UIMS's are explicitly designed to handle multiple processes. Research has shown that people can be more effective when they operate multiple input devices concurrently. It is also often easier to use multiple processes to program multiple interactions where the user can choose which interaction to use.

The disadvantages of event languages is that they are often very difficult to use to create correct code because control flow is not localized. Small changes in one part of the program can affect many other parts. It is often difficult for the designer to understand the code once it gets large.

Declarative languages state what should happen rather than how to make it happen. The interfaces supported by declarative language are usually form-based. The user types text into fields or selects options with menus or buttons. There are often graphical output areas for use by applications. The application is connected to the interface through global variables that are set and accessed by both application and interface.

**COUSIN(COoperative USer INterface)** [Hayes 85] produces an interface definition centered around form-based interface abstraction, expressed in an interpreted language. Such an interface definition consists of a declaration of the form name followed by a sequence of field definitions containing attributes. **COUSIN**'s interface definition language is based on a communication abstraction between end-user and application, in which communication takes place through a set of value-containing slots with one slot for each piece of information the end-user and application need to exchange. However, the resultant interfaces must be forms-based.

The advantage of declarative language-based UIMS's is that they free designers from worrying about the sequence of events, so they can concentrate on the information that is passed back and forth. The disadvantage is that they support only form-based interfaces. Others must be hand-coded in the graphical areas provided to applications. Also, they provide only pre-programmed, fixed kinds of interactions. For example, they provide no support for such things as dragging graphical objects, rubberbanding lines, or drawing graphical objects.

**Menulay** [Buxton 83] is an example of graphical UIMS in the second generation. A graphical UIMS lets one define the interface, at least partially, by placing objects on the screen with a mouse. The philosophy behind this approach is that, because the visual presentation of the interface is one of its most important aspects, a graphical tool is the most appropriate way to specify that presentation.

This technique is usually much easier for the designer to use. **Menulay**, for instance, can be used by non-programmers. Three disadvantages of this technique are that

- The UIMS itself is more complicated to build.
- It supports the creation of a limited range of interfaces.
- It forces the application to handle such things as help screens, aborting and prompting.

**Menulay** lets the designer place text, potentiometers, icons, and buttons on the screen and see exactly what the user will see when the application runs. Each active item in the display is associated with a semantic routine that is invoked when the user selects that item with a pointing device. Like virtually all UIMS's, the semantic routines are written in a conventional programming language. **Menulay** generates

tables and code that are linked to its run-time support package which executes the interface. **Menulay** generates its own interface and supports the concurrent operation of multiple input devices. However, its rigid table-driven structure limits the interaction between the semantic level and the interface, preventing semantic feedback.

**Syngraph (SYNtax-directed GRAPHics)** [Olsen 83] uses BNF-grammar to specify the user interface. Grammar-based systems are good for textual command languages, but they have mostly failed for graphics programs, for reasons similar to those given for state-transition diagrams.

**Syngraph** generates an interface program in Pascal from a description written in a formal grammar using an extended BNF. It handles prompting, echoing, and errors. It provides menus, textual input, and a few pre-defined interaction devices (locater, valuator, and pick) with some limited tracking. **Syngraph** can deal with semantic error recovery, Cancel and Undo at the semantic level, and deciding what to select when multiple items are on the screen at the pick position. However, **Syngraph** does not provide semantic feedback or defaults because there is no way for application routines to affect the parsing.

The UIMS's of the second generation varied vastly in their capabilities and even more in their usability. Many UIMS's were still limited in the kinds of interfaces they could produce, often covering only a small portion of possible user interface styles, techniques, and devices. Other UIMS's have greater functionality, particularly in supporting the development of rather complex graphics in the interfaces they produce. With this expanded functionality sometimes

comes a decrease in usability. Many of the UIMS's of the second generation still required significant conventional programming. Thus, as with the first generation, second generation UIMS's were often tools for an application programmer, not an interface developer.

Despite their initially limited scope -- that of execution-time for the user interface -- UIMS's near the end of the second generation began to address issues relevant to more phases of the software development lifecycle. In particular, they began to explore new techniques for specifying the interface, for example through the use of state-transition diagrams. Increased emphasis on prototyping also broadened the UIMS perspective into the software engineering world, by providing more support for this well-recognized aspect of system development. Some second generation UIMS's were actually used in commercial development environments but they were not integrated into other software engineering tools of that time. Yet even this isolated use further supported the efficiency of UIMS's as a viable tool for interface development, leading to their further research, development, and use.

c. Third Generation.

The third generation UIMS's have broadened sufficiently in their orientation so that virtually all emphasize design-time activities for user interface development [HIX 89]. In particular, they started a strong trend away from programming of the interface toward interactive, often direct manipulation tools for developing the interface. Their functionality is increasing, the kinds of interface styles, techniques, and devices that can be produced using third generation

UIMS's are expanding to include windows, mice, and other features that were rarely handled by previous generation UIMS's. In addition, the type of dialogue they address is moving away from the sequential, turn-taking kinds of dialogue that were most often addressed in previous generation UIMS's toward direct manipulation, asynchronous, and complex graphical dialogue.

Coupled with increased functionality is an attempt at improved usability. This is being achieved through the direct manipulation interfaces which many third generation UIMS's now have. More attention is being paid to the user interface of the UIMS's themselves, and their interfaces are now beginning to be both empirically developed and evaluated. Improved usability is also being achieved by different approaches to the UIMS interface, such as "by demonstration"

Approaches to building a UIMS have also changed during the third generation, with many UIMS's now being built using an object-oriented paradigm or using a windowing package such as X-windows.

**Peridot (Programming by Example for Real-time Interface Design obviating Typing)**[Myers 87], **Trillium** [Henderson 86], **Grins** [Olsen 85], and SmetherBarnes **Prototyper** [Prototyper 87] are all graphical-based UIMS's.

**Trillium,** which is very similar to **Menulay**, supports the design of interface panels, for photocopiers. One strong advantage **Trillium** has over **Menulay** is that it interprets rather than compiles the specification, so the frames can be executed as they are designed. **Trillium** also separates the interaction behavior from the graphical presentation so the designer can change the graphics without changing

the behavior. However, **Trillium** provides little support for frame-to-frame transition because that is rarely necessary in photocopiers.

**HyperCard** hypertext system belongs in the graphical UIMS class. It supports graphical specification (and programming in the HyperTalk language) of mostly static pages. Using the editor, the designer can define the text and graphics for the current page, and buttons that cause transitions to other pages.

**Grins** combines a grammar processor with a constraint-based, input-output linkage system to handle semantic feedback. It incorporates a graphics editor that lets the designer place interaction techniques (menus, icons, and text areas) with a mouse.

**Peridot** is very different from these systems because it lets the designer create the interaction techniques themselves. It uses a "by demonstration" mechanism for interactively developing the user interface. The interface developer represents how input devices are to be handled by showing examples of their use. The designer manipulates primitives (rectangles, circles, text and lines) to construct menus, scroll bars, sliders, (graphical potentiometers) and buttons. Peridot generalizes from the designer's actions to create parameterized, object-oriented procedures like those found in interaction technique toolkits. **Peridot** can be used to represent devices found in direct manipulation interfaces, including mouse and touch tablet.

The SmetherBarnes **Prototyper** is a commercially-available tool that, despite its name, is more a UIMS than a prototyper. It can be used in a direct manipulation fashion to develop Macintosh-style interfaces, including windows, pull-down menus, radio buttons, check boxes and other objects typically found in Macintosh application interfaces. No

programming is required to produce the interface. The application semantics can be coded in one of several programming languages and linked to the interface for execution.

An object-oriented language-based UIMS provides an object-oriented framework in which the designer programs the interface. Typically, objects from classes handle default behavior. The designer specializes these classes to deal with behavior specific to the interface using the inheritance mechanism built into object--oriented languages. **MacApp** [Schmucker 86] is programmed in Object Pascal. **GWUIMS** [Sibert 86] uses object-oriented Lisp and provides a classification of interface operations and objects that fit into each class. **Higgens** [Hundon 86] adds a structured data description that supports Undo and Redo and lets the UIMS automatically manage the recalculation and re-display of objects intelligently.

Object-oriented systems can handle highly interactive, direct manipulation interfaces because there is a computational link between the input and the output that the application can modify to provide semantic processing. Although these systems make it much easier to create interfaces, they are programming environments and as such, inaccessible to non-programmers.

The third generation of UIMS started a revolution. Recognition of the importance of the user interface is now well-accepted and therefore the need for tools to support development for the interface is well-motivated. The increased functionality and usability of third generation UIMS's have allowed them to reach even further into the software engineering realm, and to support even more phases of the software engineering life cycle. In particular, interface development is becoming

espoused as an integral and equal part of the entire software development process.

The traditional linear waterfall model is no longer the accepted approach to developing the interface. Rather, an evaluation-centered lifecycle is emerging as a more natural and useful paradigm for producing quality user interfaces. This has led to integrating these tools with CASE tools. Sophisticated techniques for specification of a broader range of interface types, in an easier manner with increased functionally and usability, have caused UIMS users to realize their enormous potential. This is especially true when a UIMS is integrated into a software engineering development environment, giving interface development an equal emphasis with the rest of the application system.

Present day UIMS's are well into the third generation and moving toward fourth generation. It is in fact the success of this generation of UIMS's that will determine if they are to become an accepted, viable tool, moving them from the realm of research into the real world of software development.

d. Fourth and Future Generations.

We shall speculate about fourth and future generations, based on the trends we perceive in UIMS development. Researchers and practitioners in the field of user interface development readily espouse a development approach based on the concept of evaluation and interactive refinement of the interface. Now it is time to apply this development approach to the production of UIMS's themselves.

Fourth and future generation UIMS's will have improved usability, often as a result of empirically-based derivation, iterative refinement,

and enhancements beginning to be found in real world user software development environments.

## C. Ultimate goal of UIMS

Hix[Hix 89] states that the ultimate goal, of course, is to have UIMS's integrated into the software development environment, supporting all aspects of user interface production throughout the complete lifecycle. Every phase of the interface development lifecycle can potentially and should be supported by UIMS tools.

Rapid prototyping is key at almost every phase of the interface development lifecycle; during task analysis some early design work is typically done, for example, use of informal screen sketches and experimentation with various interaction styles. When these doodles are captured interactively through a UIMS, they can be used as the vary earliest prototypes of an interface, and can be carried forward to subsequent development activities, being refined as evaluation occurs. Rapid prototyping is indispensable in design and pre-implementation lifecycle activities, when much more of the interface, in a more concrete form, can be made available for end-user testing and iterative refinement. Support of evaluation efforts after implementation includes the need for tools to capture log data of end-user activities during interaction with the system, as well as to expedite the collapse and analysis of these data.

In summary, a UIMS needs to support usability engineering, to allow visibility of interface behavior, and to provide rapid modifiability, accommodating easy and effective evaluation and sometimes massive design changes throughout the lifecycle.

When support for each phase of the interface development lifecycle is available, UIMS's may become ubiquitous and as indispensable as tools such as database management systems, language compilers, and programming environments are currently. They will be fully integrated into CASE-like environments that give recognition to the importance of development of the user interface. Only then will we be able to take advantage of the potentially vast improvement in productivity that can be achieved through use of UIMS.

By developing better tools for producing user interfaces, we can expect, ultimately, to produce better user interfaces. We have much to gain from continued pursuit of this exciting and fast-growing field. This exploration will take us to the next generation of UIMS and beyond.

# II. Oregon Speedcode Universe (O.S.U.)

## A. Integration of UIMS and CASE

O.S.U. combines a UIMS with a structured design facility which allows a programmer to quickly prototype the user interface of a given application and then connect that interface to program design tools traditionally found in most CASE systems.

The UIMS allows a programmer to create and directly manipulate icons, menus, windows, dialogs, alerts, and user-defined procedures. It consists of three parts: 1) a resource editor called RezDez (Resource Designer) for WYSIWYG creation and editing of icons, menus, windows, dialogs, and alerts; 2) a graphical sequencer, which allows a user to "train" the application to behave the way it should when the application is used by an end user; and 3) a program generator that writes a Pascal source code program for implementing the behavior specified by "sequencing". For a complete discussion of UIMS and rapid prototyping in O.S.U., see [Lewis 89].

With its UIMS alone, O.S.U. is able to create the complete user interface portion of the application. In most other UIMS's, functions are added by writing code in the traditional manner. O.S.U., however, has chosen CASE as the approach to add functionality to applications.

CASE is the automation of software development. The basic idea behind CASE is to provide a set of well-integrated tools, linking and automating all phases of the software lifecycle. Different CASE tools focus on the support of different phases of the software lifecycle or on the development of different types of software systems. There are 3

basic categories of CASE tools: CASE toolkits, CASE workbenches, and CASE methodology companions [McClure 89].

CASE toolkits are a set of integrated tools that automate one type of software life cycle task, such as system design, program maintenance, or one type of job class, such as system analyst. A dataflow diagramming tool, for instance, is an example of a system analysis toolkit.

CASE workbenches are composed of a set of integrated tools that automate tasks across the entire software lifecycle, namely, analysis, design, and implementation. The tools are integrated to the level that the output of one lifecycle phase is directly and automatically passed on to the next phase. The final product of a CASE workbench is an executable software system and its accompanying documentation.

A CASE methodology companion is either a CASE toolkit or a CASE workbench that structures the software development process according to the steps and rules of a particular structured methodology. Information about the methodology is embedded into the methodology companion by means of help panels, menu choices, forcing functions, and quality assurance checks.

CASE tools provided by O.S.U. are design/implementation toolkits integrated together as one. The Structure Chart Editor and VIGRAM are modular design and detailed design tools used to design user-defined procedures/functions as well as to access reusable software components. Unlike most CASE design toolkits, Structure Chart Editor and VIGRAM are also implementation toolkits in that they generate Pascal source code automatically from design specifications.

## B. Components of O.S.U.

### 1. User Interface Management System (UIMS)

#### a. RezDez

RezDez ( Resource Designer ) is used to graphically create and edit all user interface objects -- menus, icons, dialogs, windows, alerts, error messages, prompts, and associated information [BOSE 88]. These objects are "painted" on the screen exactly as they initially appear in the finished application. Figure 2 illustrates how a dialog is created by direct manipulation of its size, shape, and items.

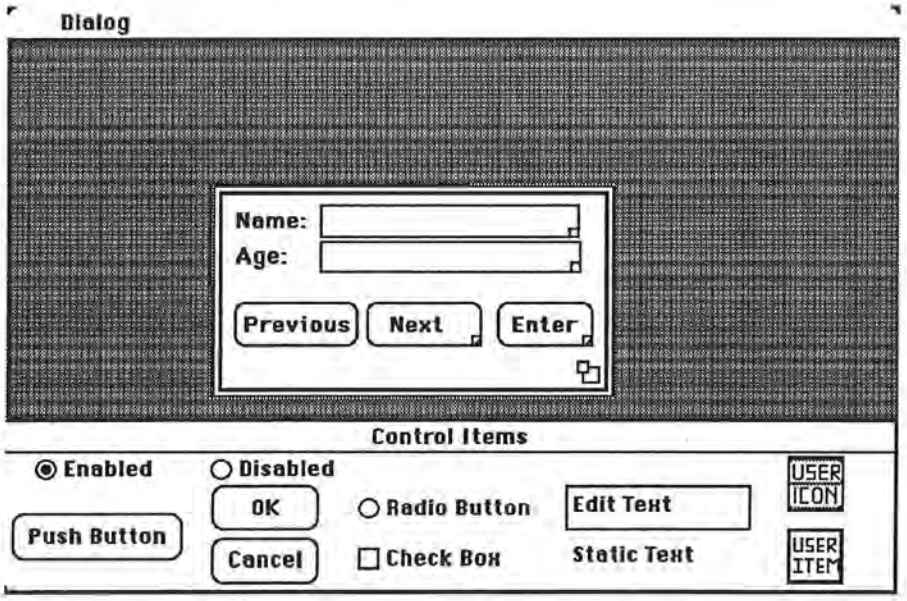



**Figure 2. A Dialog in O.S.U. is created by dragging items from the tool palette onto the new dialog.**

RezDez not only creates each object, but it also defines the initial internal state of the object. The description of the object, in its initial state, is stored as a separate *resource* in the application's binary file called the *resource fork.*

b. Graphical Sequencer

The graphical sequencer is used by a programmer to specifying the sequencing information of the user interface objects created in RezDez. It allows the programmer to "play out" the application by doing rather than writing instructions in the form of a script or textual language.

When the prototype under construction is shown in action (simulated), such as pulling down a menu to make a menu selection, the graphical sequencer "calls" the appropriate behavior defined for the menu. The behavior carries out the operation, thus changing the state of the object, and the configuration of the user interface. Figure 3 shows a directed graph representation for a simple sequence involving a menu selection and two dialogs.

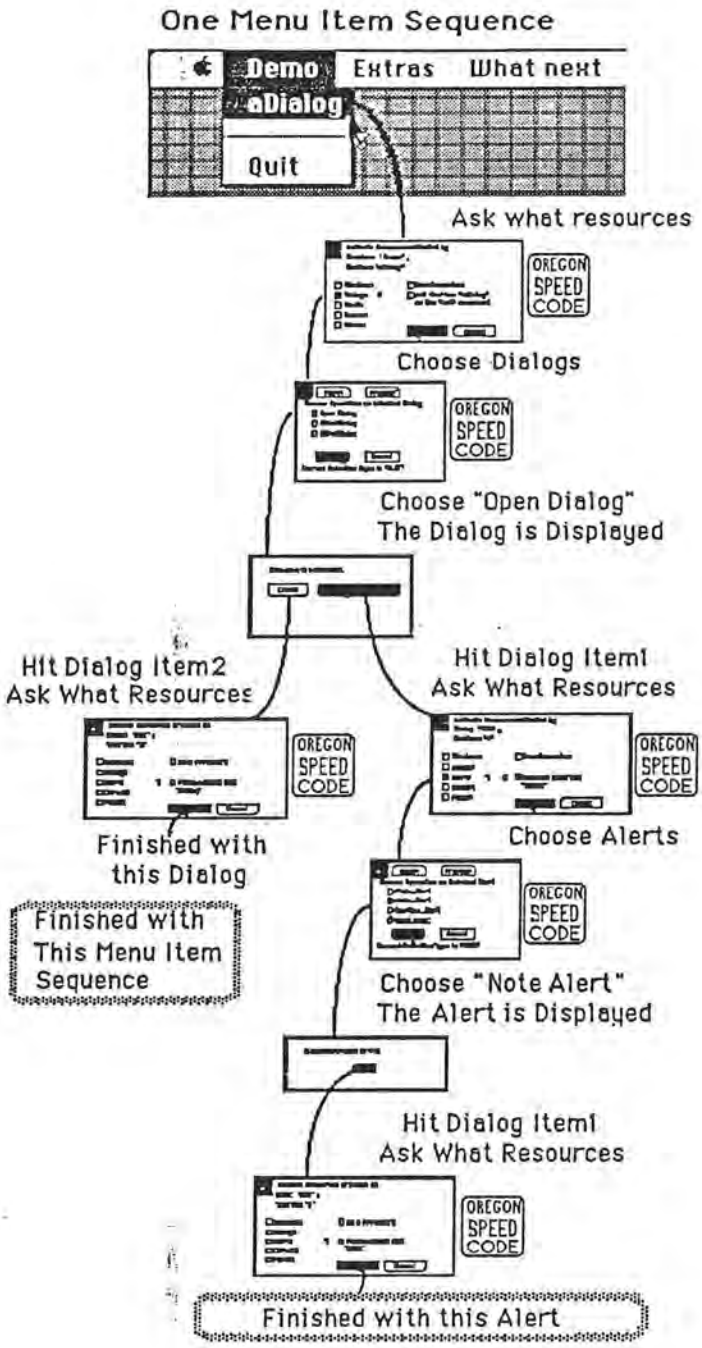



Figure 3. An Example of a Sequence from the Graphical Sequencer of O.S.U.

c. Code Generator

Using the sequencing information gathered by the graphical sequencer, the code generator is able to generate user interface code automatically. The sequencing information is stored in a sequence command language format [Armstrong 88]. The source code generated is in Pascal, and can be complied and linked by the Lightspeed Pascal compiler.

## 2. CASE tools

a. Structure Chart Editor

A structure chart is a tree or hierarchical diagram that defines the overall architecture of a program through its *call graph* [DeMarco 78]. Structure charts are the graphical representation of functional decomposition. The basic building block of a program is a module, and structured programs are organized as a hierarchy of modules.

It is important to distinguish between a program module and a program function/procedure. A module is a separately compiled unit of code consisting of function/procedure definitions along with interface specifications. In O.S.U., the separate compiland called a Macintosh Pascal *unit* is used to form modules. Every unit consists of an interface part and an implementation part, very similar to modules in Modula-2, and packages in Ada™.

A structure chart is a call graph showing the interconnections among procedure/functions. Interconnection is shown by arranging procedure/functions in levels and connecting them by arcs. An arc drawn between two modules at successive levels means that at execution time program control is passed from one module to another in

the top-down direction. Modules are not procedures, and because a structure chart only shows procedure/function interconnection, the structure chart is only one of several architectural points of view. We can, for example, render a program in terms of other points of view, such as its *uses relation*, which consists of a graph showing what units are used by each unit in the program or an object-oriented rendering showing both call graph and uses relations.

In O.S.U., rectangular boxes represent procedure/functions and arcs connecting the boxes represent invocations or calls to each procedure/function. The unit name and procedure/function name is displayed in each box to allow combination of functional decomposition and object-oriented design methodologies.

The structure chart in Figure 4 shows a call graph for a file input routine. All the file related routines are grouped in a unit called FILE. Macintosh provides some standard file routines for opening, reading and writing files. All of the Macintosh ROM-based routines are collectively called the TOOLBOX routines. ErrorCheck is a reusable routine from the GrabBag reusable component. DIA_showAlert is a general alert routine in the DIALOG unit, which is called in case of a file open error.

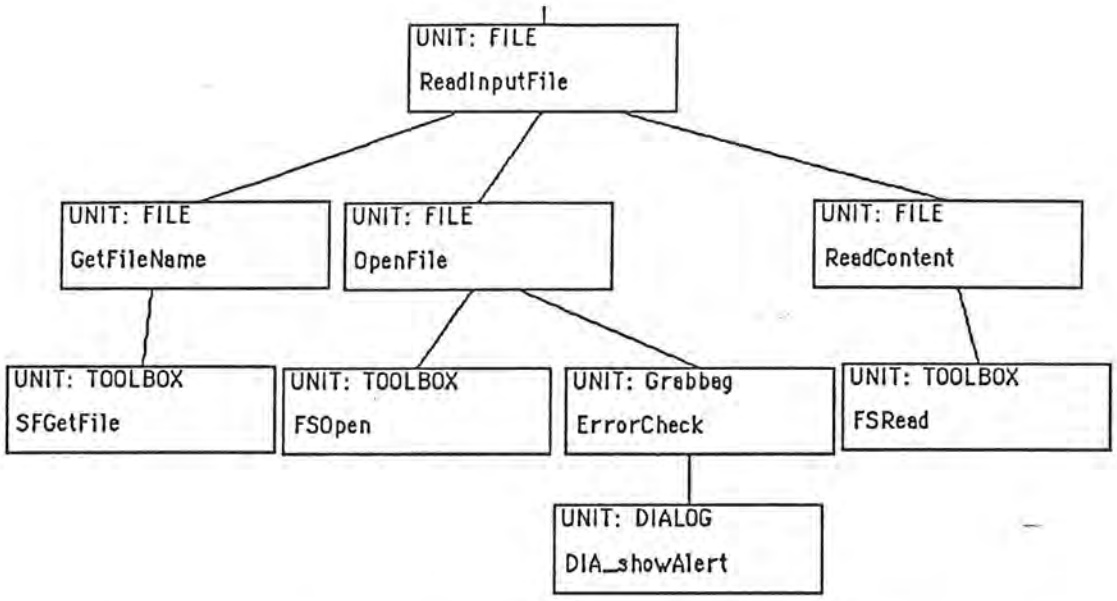



**Figure 4. A sample structure chart created by O.S.U..**

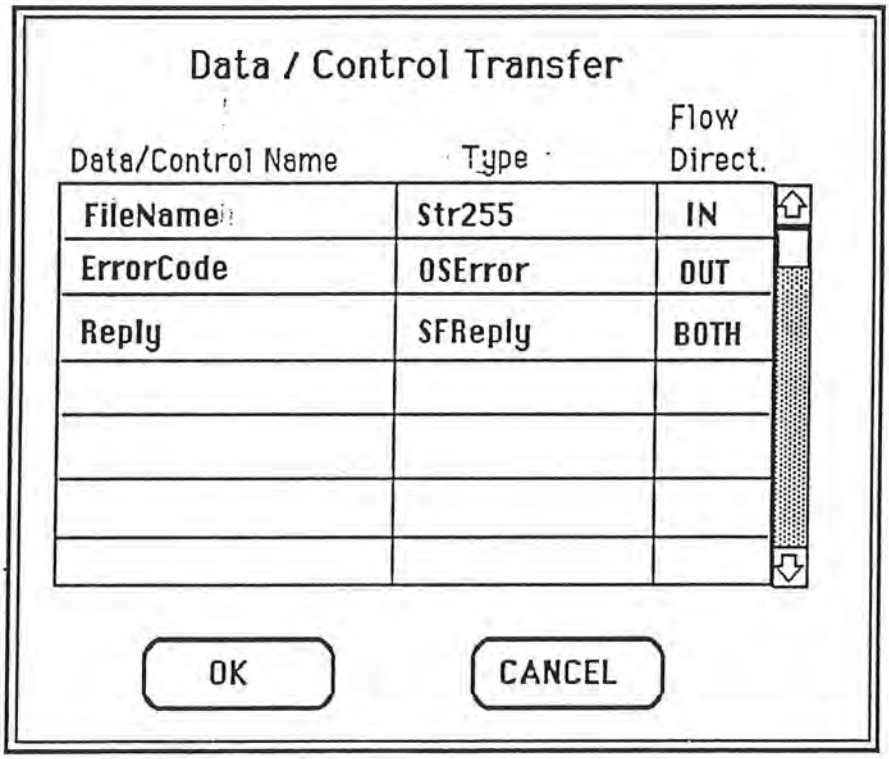



**Figure 5. A dialog for displaying data/control transfers between procedure/functions.**

Data and control (flags, error code, etc.) transfers between procedure/functions are usually in the form of parameter passing. Conventional methods of displaying the structure chart draws the parameters next to the arc that connects the two procedure/functions, but for the purpose of readability, O.S.U.'s structure chart editor hides the details of the parameters. The programmer must double-click the arc to display the parameters as shown in Figure 5. This dialog displays parameter information in the form of data and control flow in and out of the function/procedure. For each data item, the name, data type and flow direction is required. **IN** flow means the information is coming into the module. **OUT** flow means the information is returned from the module. **BOTH** means the information is both coming in from and passed back to the calling module. In generating PASCAL source code, **IN** data are the value parameters, **OUT** data are the variable parameters that do not need to be initialized upon calling the procedure, and **BOTH** data are the variable parameters that need to be initialized before calling the particular procedure.

- Object-oriented view.

Object-oriented development is an approach to software design in which the decomposition of a system is based on the concept of an object [Booch 86]. An object is an abstract data type entity with the ability to inherit properties from classes of other objects. An object has state and function -- state in the form of data, and function in the form of function/procedures. In Macintosh Pascal, an object is defined as a *unit*. Units cannot inherit functions from other units, so our approach is not pure. Instead, units are used to encapsulate state in the form of

constants, types, and variables, and function in the form of functions and procedures. An object hierarchy is established as a uses graph -- one more architectural point of view of interest to us.

Syntactically, Pascal units are connected by the "uses" clause which is a mechanism for import/export of constants, types, variables, procedures, and functions that are visible from outside of a unit. Thus modules are connected via their interface parts and access procedure invocations.

Rather than factoring the system into modules that denote operations, we structure the system around its objects, or units. Each object is represented as a rectangular box with its name at the top, and all operations defined on the object are listed within the rectangle. Interconnections between objects are shown as arcs and represent function invocation just as in the structure chart view. Objects are arranged hierarchically based on their uses relations.

Figure 6 shows the object-oriented representation of the ReadInputFile procedure of Figure 4. There are 4 units, FILE, TOOLBOX, GRABBAG, and DIALOG. Units are arranged in a hierarchical fashion. TOOLBOX and GRABBAG units are directly called from the FILE unit, so they are arranged 1 indentation from the FILE unit. The DIALOG unit is called from the GRABBAG unit, so it is arranged 1 indentation from the GRABBAG unit. The visible procedure/function names of each unit are displayed in the small rectangle inside each unit.

This representation differs from other proposals for displaying object-oriented designs [Booch 86]. In particular, our representation does not distinguish between an object and its class. In object-oriented terminology, our model does not show instantiated classes, but instead,

shows only the concrete objects actually used. This is partly a result of weaknesses in Pascal, and partly our desire to simplify the representation.

Why is the object-oriented view important? According to Booch [Booch 86], the object-oriented approach to design mitigates weaknesses in functional decomposition such as lack of data abstraction and information hiding. Object-oriented design is generally thought to be more responsive to changes in the problem space.

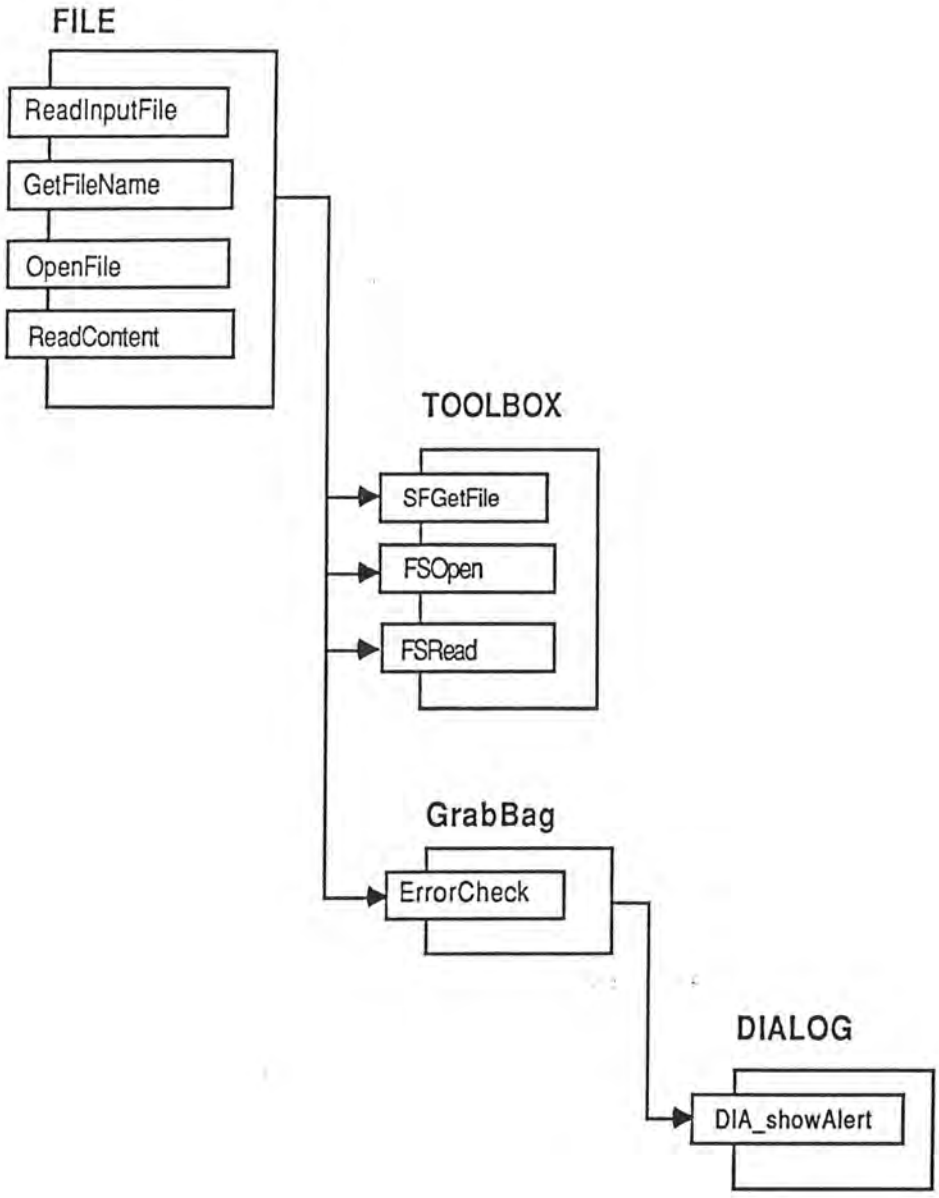



**Figure 6. An object-oriented view of Figure 4.**

- Uses Graph.

The problem with object-oriented design is coming up with an object-oriented rendition of the entire system. Unlike other forms of representation, object-oriented designs do not incorporate hierarchical structure. To reduce the clutter of an un-leveled object-oriented view, a simplified uses graph can be generated as shown in Figure 7. The uses graph suppresses the connections in the system stemming from calls and shows only the import/export properties of the objects.

The graph in Figure 7 shows only the *uses relation* among units. Again the units are arranged in an hierarchical fashion based on their uses order.

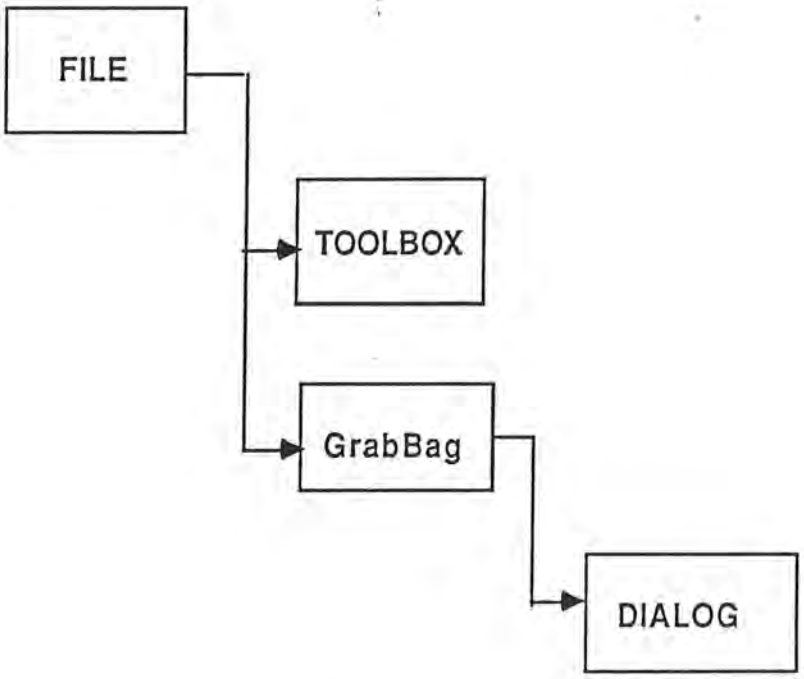



**Figure 7. Unit Uses Graph for Figure 6.**

b. VIGRAM

A major drawback of the structure chart method as a design representation is that control structures such as repetition, sequence, and alternation are not easily represented. Instead they are regarded as details which are shown by a different technique, such as pseudocode, flowcharts, etc. Such details can be rendered by a *detailed design tool.*

Detailed designs are visually constructed in O.S.U. using VIGRAM, which is a graphical tool for editing Pascal source code [Hsieh 88]. A procedure or function can be created by either reading an existing procedure of function from a reusable unit and modifying it, or by creating an entirely new unit. In either case, the programmer designs the procedure from visual building blocks. See Figure 8.

Once the detailed design specification is provided, Pascal source code is automatically generated from VIGRAM. The VIGRAM serves also as an understanding tool and a maintenance tool through two techniques: Program slicing and complexity metrics [Yang 89].

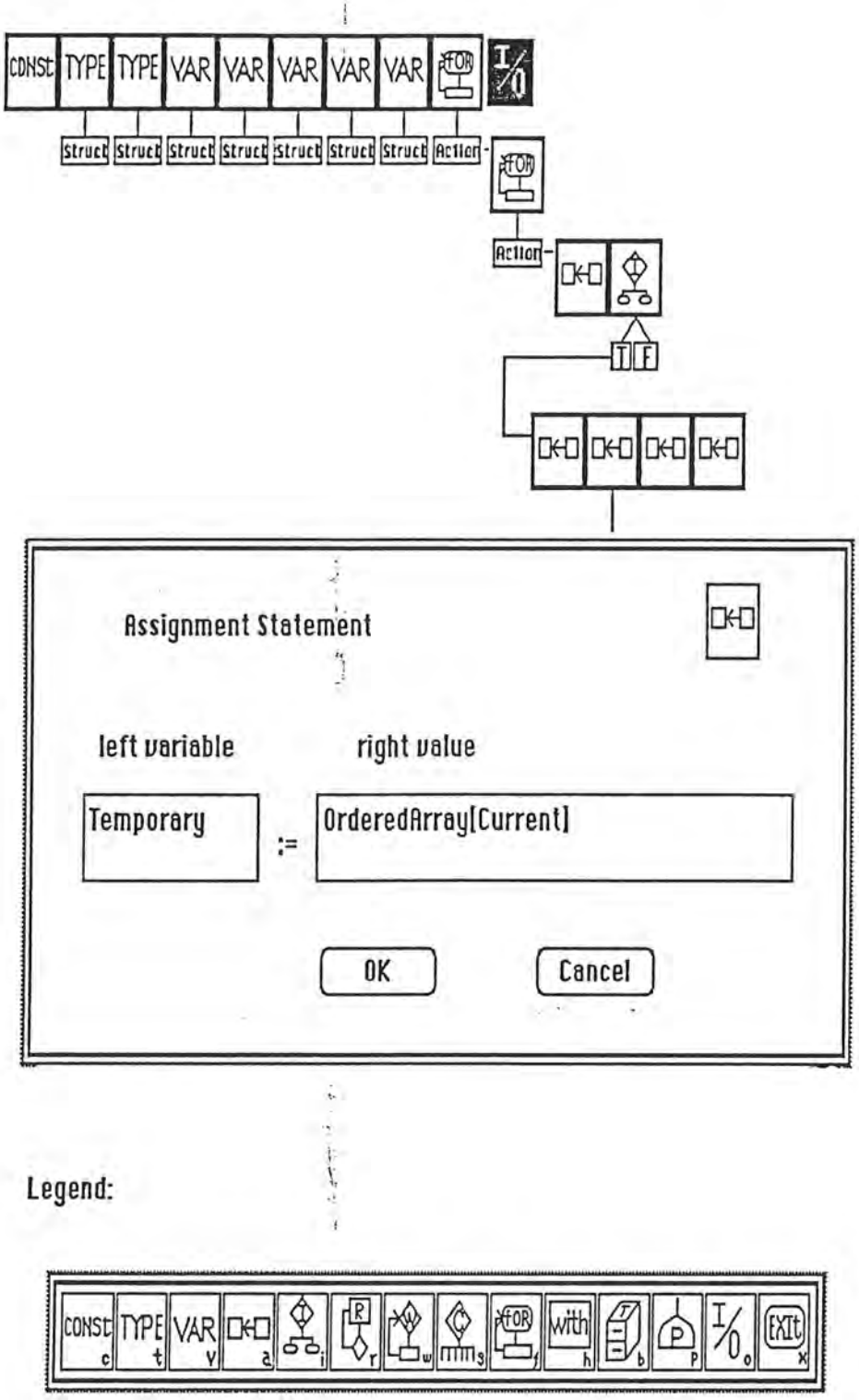



Figure 8. VIGRAM view of a Bubblesort procedure.

## 3. Software Accelerators

Current work is underway in design and implementation of domain-specific tools called *software accelerators*. These software accelerators will allow more functionality to be added to resulting applications. Areas under investigation are: database, data structures, graphics, text, mathematics, sound and animation [Lewis 89].

# III. Design of Structure Chart Editor

Three design decisions had to be made before the Structure Chart Editor was implemented: 1) How to integrate structured design methodology with user interface design, 2) How to create and reuse software components, and 3) How to combine functional decomposition with object-oriented methodology. These decisions were influenced by the development environment and programming language used, namely, Think Technology's **Lightspeed Pascal.**

## A. Integration of UIMS and CASE

The Structure Chart Editor can be combined with UIMS by integrating the user interface specifications with the functional specifications as follows. All user interface objects are designed and stored along with the "sequence" information needed to make the user interface "simulate" the final application. (See Section II). These objects and their sequence information are called a *user interface prototype* in O.S.U., because they specify the user interface, but none of the application's functionality.

In the process of sequencing through the user interface, functionality of the application can be added in the form of:

1) **User procedures.** The Structure Chart Editor will be invoked from the Graphical Sequencer to allow a new code produced by hand, or reusable components taken from a library of reusable modules, or Macintosh **toolbox** calls to be added to the system.

2) **Software accelerator reusable components.** The extended graphical sequencer will handle modules which are automatically produced by one of several software accelerators [Raghu 89].

As shown in Figure 9, when the programmer selects **Do a Procedure** in the graphical sequencer, control is transferred from graphical sequencer to the structure chart editor. A structure chart window comes up along with the structure chart menu. The programmer is asked to define the complete modular structure of the user structure.

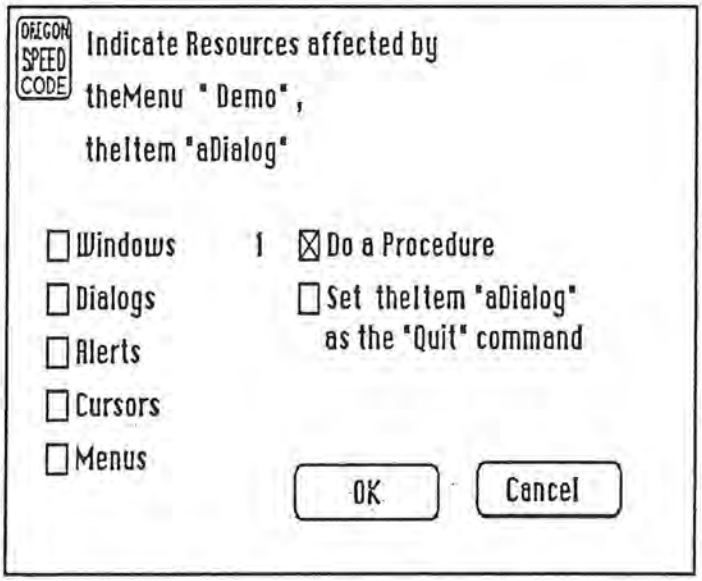



**Figure 9. Selecting of DO procedure from the graphical sequencer**

The newly inserted procedure/function is called an *active object* in O.S.U., and is added to the sequence in the form of a **DO** routine. The sequence of user interface interactions and active objects is recorded and written in a *sequence language script*. When the Pascal source code is generated from the recorded sequence language script, a place-holder is inserted for each of the programmer-defined active objects.

All **User procedures** are combined into one structure chart to allow the user to view the entire modular structure of the whole application.

## B. New and Reusable Software Components

In order to create a new module or reuse a reusable component, the Structure Chart Editor must be integrated with VIGRAM. Because neither alone provides enough detail for implementation.

To create a new procedure, the user first uses the Structure Chart Editor to layout the modular structure or call graph of the new procedure. VIGRAM is then called to specify the detail of each structure chart box. Pascal source can be generated automatically, because VIGRAM specifies detail down to the statement level.

In order to reuse a component, VIGRAM is called to read and parse the source code of the reusable component, extract the modular structure of the component, and display the component in VIGRAM's graphic format as shown in Figure 8, for ease of understanding [Yang 89]. In addition to the visual display, VIGRAM also computes a number of metrics: Barry-Meekings, Halstead, and LOC. These are related to the "complexity" of the component, and may be used to related to the complexity of the component and may be used to understand and modify the component.

The modular structure information that VIGRAM extracted will be passed to and displayed by the Structure Chart Editor.

## C. Combination of Functional decomposition and Object-oriented Methodology

In Lightspeed Pascal, objects are units. The structure chart of functional decomposition provides both the unit name and routine name as seen in Figure 4. Thus, to render the object-oriented view requires simply a re-construction of the structure chart view by grouping all routines from the same unit together. Similarly, construct its uses relations from the call structure. Significant amount of error checking is required to detect errors such as circular uses relations or circular call references.

# IV. Evaluation & Enhancements

## A. Evaluation and Enhancements of Main User Interface.

### 1. Menu.

a. Old menu.

The original O.S.U. menu, as shown in Figure 10 contained items that were unclear or unused by the O.S.U. application, thus changes were necessary for the purpose of clarification. For instance, all items under **File** and **Edit** were unused by the system except for the **Quit** option. Items in the **Speedcode** menu were unclear as to what each one does. In particular, the **Create Resources on Screen** carried 2 meanings. It served both as a new and an open option depending on whether the resource file already existed or not. **Prototype an Application graphically** performed the same operation as **Explore Graphical Sequencer**, except in the first case, the source code was generated automatically, and in the second, code generation was not performed.

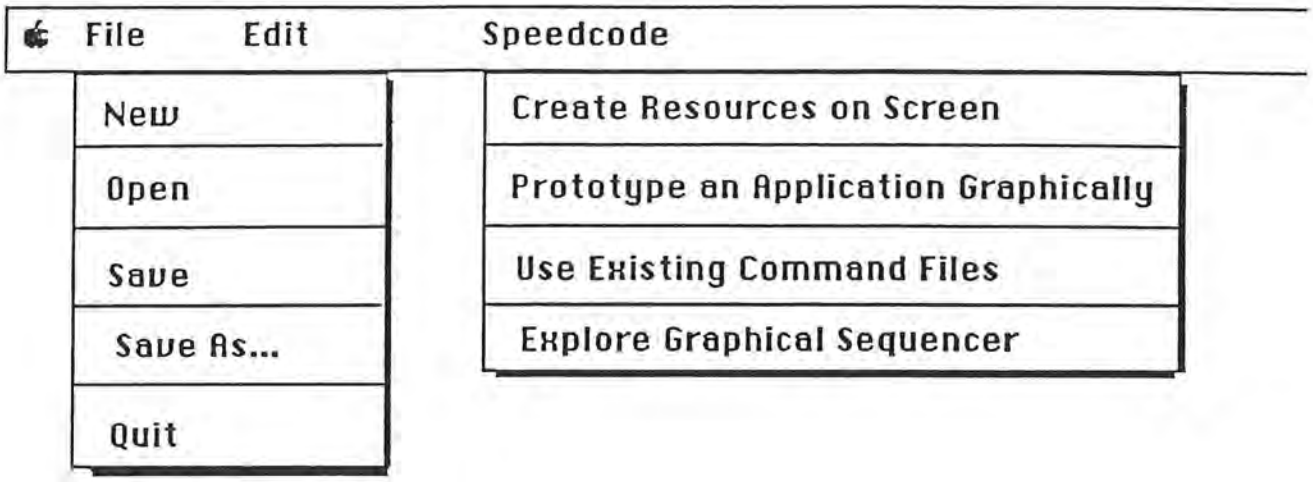



Figure 10. The original O.S.U. Main menu.

b. New menu.

In order to clearly separate prototype operations from resource file operations, the new menu bar separates the two into 2 separate menus.

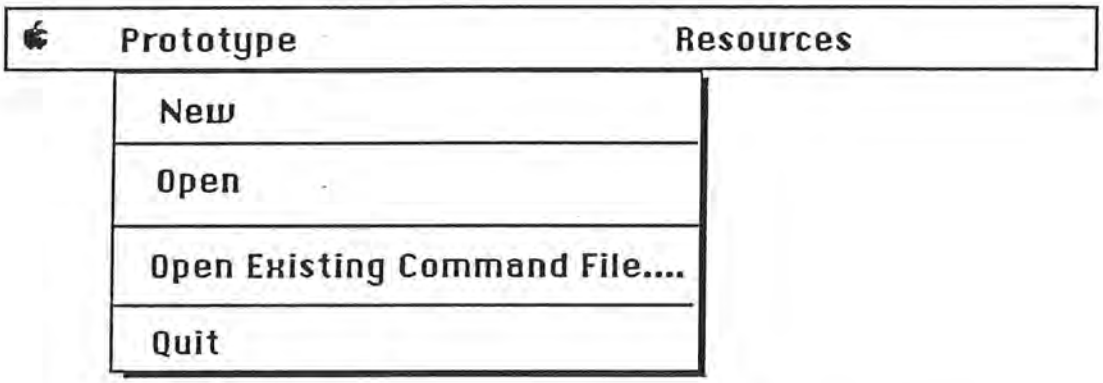



Figure 11. The new O.S.U. Prototype menu.

Under the **Prototype** menu (Figure 11), the New item will start a new prototype. It will allow the user to select a resource file and go into graphical sequencer for specifying the sequence of resource objects.

The user will be asked to save the prototype's sequencing information, and/or generate the source code before leaving the graphical sequencer. See [Chia 89] for implementation details.

The **Open** item allows the user to load and work on an existing prototype (see Functionality below). The **Open Existing Command File....** item automatically generates Pascal code from sequencing information previously saved. The **Quit** item exits the O.S.U. application.


**Figure 12. The new O.S.U. Resources menu.**

The new **Resources** menu (Figure 12), clarifies creating a new resource file vs. editing an existing one. The **Create** item creates a new resource file with .RSRC file extension as required by Lightspeed Pascal, and the **Edit** item allows an existing .RSRC file to be read and modified. Any existing .RSRC file can be read regardless of whether it was created by O.S.U. or not.

## 2. Functionality

The original O.S.U. did not provide ways to save and load a previously prototyped application. The only way to change an application was to specify the whole application over again. Therefore, the ability to save and read in an existing prototype is an essential enhancement [Chia 89].

# B. Evaluation and Enhancement of RezDez

The original RezDez [Bose 88] had several shortcomings as well as nonstandard Macintosh conventions.

## 1. Main Selection dialog

a. Old main selection dialog.

The original main selection dialog with a radio button for each resource object type is awkward (Figure 13). It required the user to make 2 clicks in order to make a selection. A better method of interaction is desired. Also as the system grows, the need for more resource types also grows with it. Therefore a new selection dialog is needed.


**Figure 13. Original RezDez main selection dialog.**

b. New main selection dialog.

The new main selection dialog is shown in Figure 14. The radio button selection technique is replaced by the more convenient and standard one-step push button technique. Additional selection choices are added to include resource types, such as cursors, pictures, and design of graphical palettes. Notice that **Icon** and **Icon and Mask** are collapsed as one in the new dialog. The distinction between and icon and an icon with a mask is made in the icon editor itself rather than at the main level.

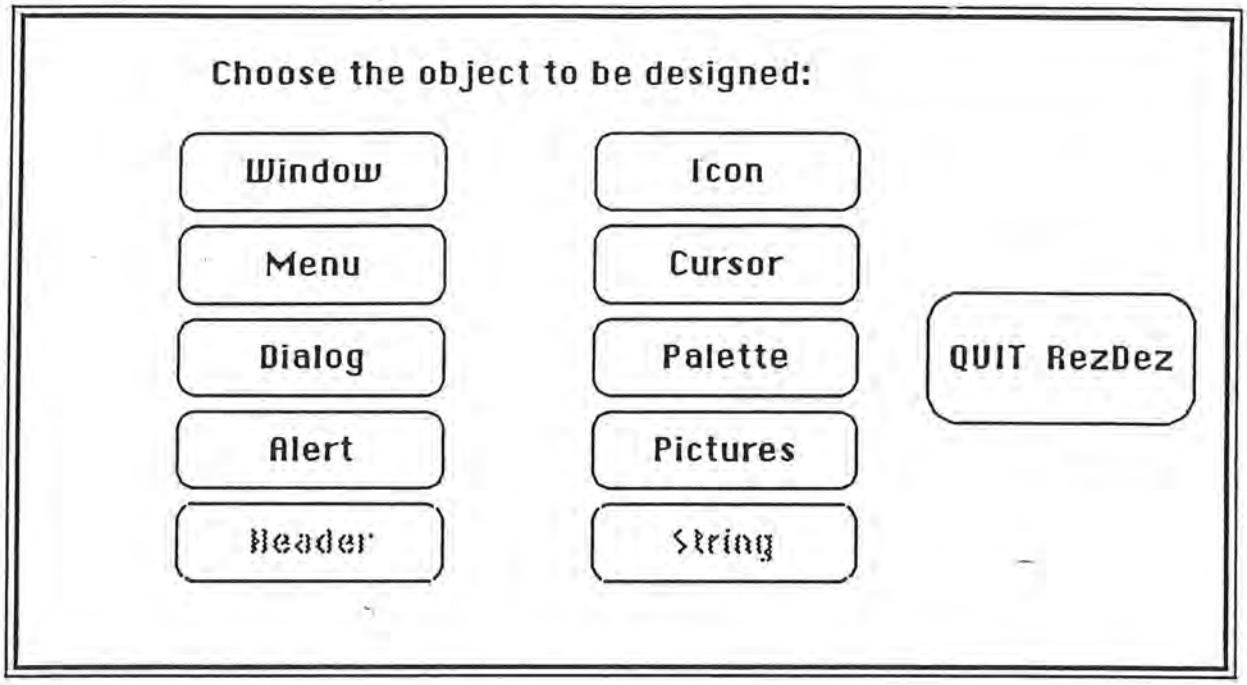



**Figure 14. New RezDez main selection dialog.**

## 2. File I/O

Original RezDez made no distinction between create and edit files because of the ambiguity in the main O.S.U. menu (See Section IV-A-1-a). This is corrected along with the main menu change. In keeping up with the new standards of Macintosh programming, RezDez was made HFS (Hierarchical File System) compatible.

## 3. Menus in each resource editor.

a. Old menu

The menu in each of the individual resource editors did not follow the standard Macintosh guidelines in that it provided no support for Desktop Accessories (DA's). Figure 15 is an example of the menu bar for the window editor.

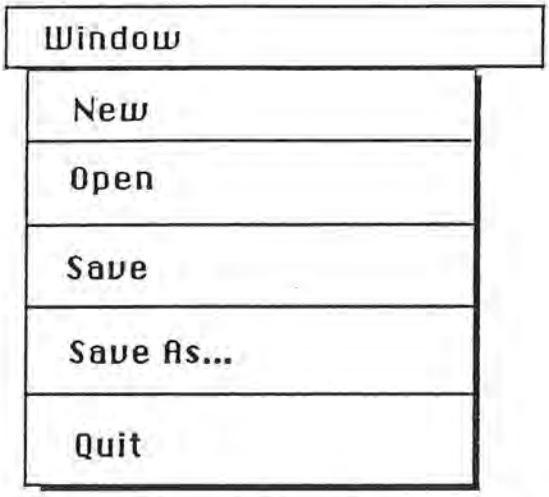



**Figure 15. Original RezDez menu for window editor.**

There are other shortcomings in this menu setup. First of all, deleting or disposing a user interface object was impossible once the object was created. Second, the **Open** item is ambiguous. There are 2 ways that RezDez can open a resource: from the resource file that RezDez is currently working with, or from other applications and resources. Distinctions should be made between these two opens to avoid any confusion for the user.

b. New menu

The standard file menu in Figure 16 clarifies the RezDez interface. This standard menu is used for all resource types, thus eliminating the need for 6 or 7 different menus for each resource type.

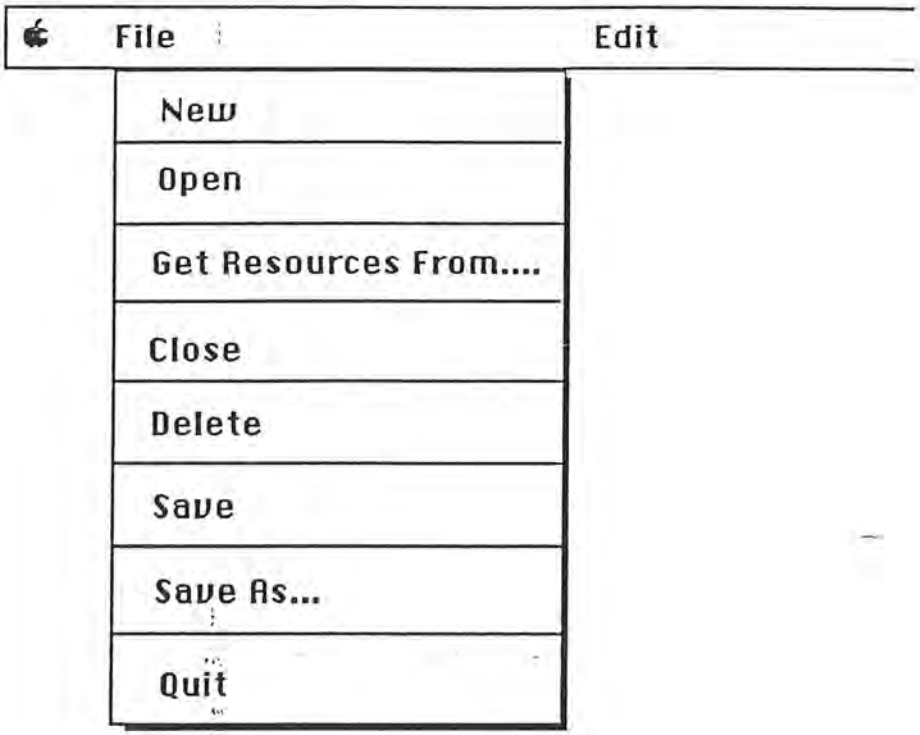



**Figure 16. New RezDez standard File menu.**

Under the **File** menu, the **New** item allows the user to create a new resource object; **Open** allows the user to get an existing resource item from the current resource file that RezDez is working with; **Get Resources from....** allows the user to get existing resources from other applications or resource files, i.e. reusable resources. **Close** closes the current resource while **Delete** removes existing resources from the current resource file. **Save & Save As...** saves the resource in the resource file and **Quit** will take the user out of the resource editor, and return to the main selection dialog.

## 4. Preview capability

The original RezDez did not allow previewing of a resource object before it opened it. This proved to be a limitation, because often times, user can't correctly identify the resource items desired simply by its resource ids.

The new RezDez allows the resources to be accessed by id or name. The preview capability allows the resource items to be previewed before they are opened. (See Figure 17). Preview means the resource is displayed on the screen before it is read into RezDez.

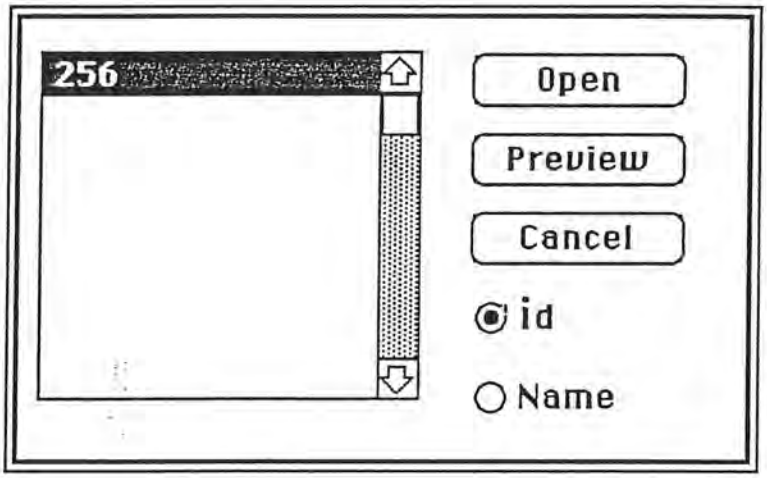



**Figure 17. RezDez Open Resources Dialog.**

## 5. Enhancements to Windows.

RezDez's window editor didn't handle the zoom box option for document windows. Figure 18 shows the new window info dialog that has been added to allow the user to specify whether a goAway and/or zoom box are desired for a particular window. The window info dialog is displayed when the user double clicks in the window's content region.

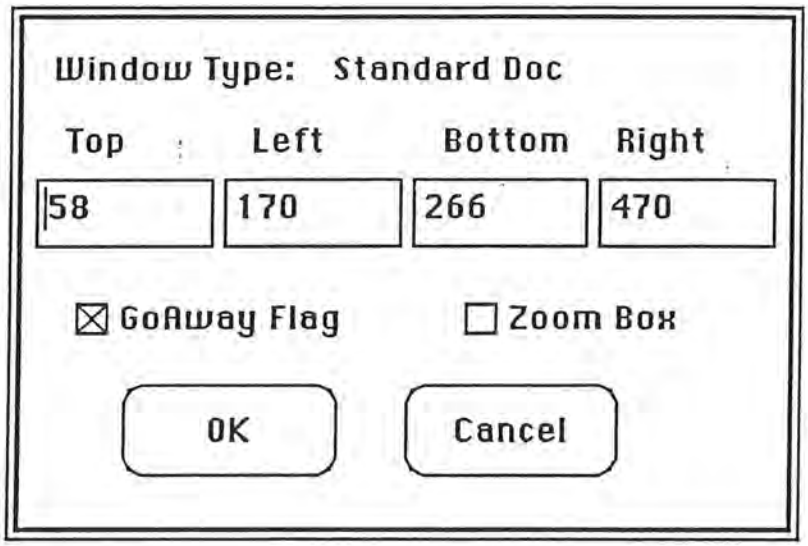



Figure 18. RezDez's Window Info Dialog.

## 6. Enhancements to Menus

a. Old menu entry dialog.

The original RezDez's menu entry dialog was not user-friendly (Figure 19). RezDez forced the user to perform certain actions in an order that the user could not control. For instance, in order to add a keyboard equivalent to a menu item, the user was forced to select the **keyboard equv** option in the option list, push the **Add Option** button immediately followed by typing a character in the item name box, followed by clicking the small **ok** button appearing at the bottom of the option list. This is not only awkward, but it takes control away from the user. Therefore, a better user interface is needed.

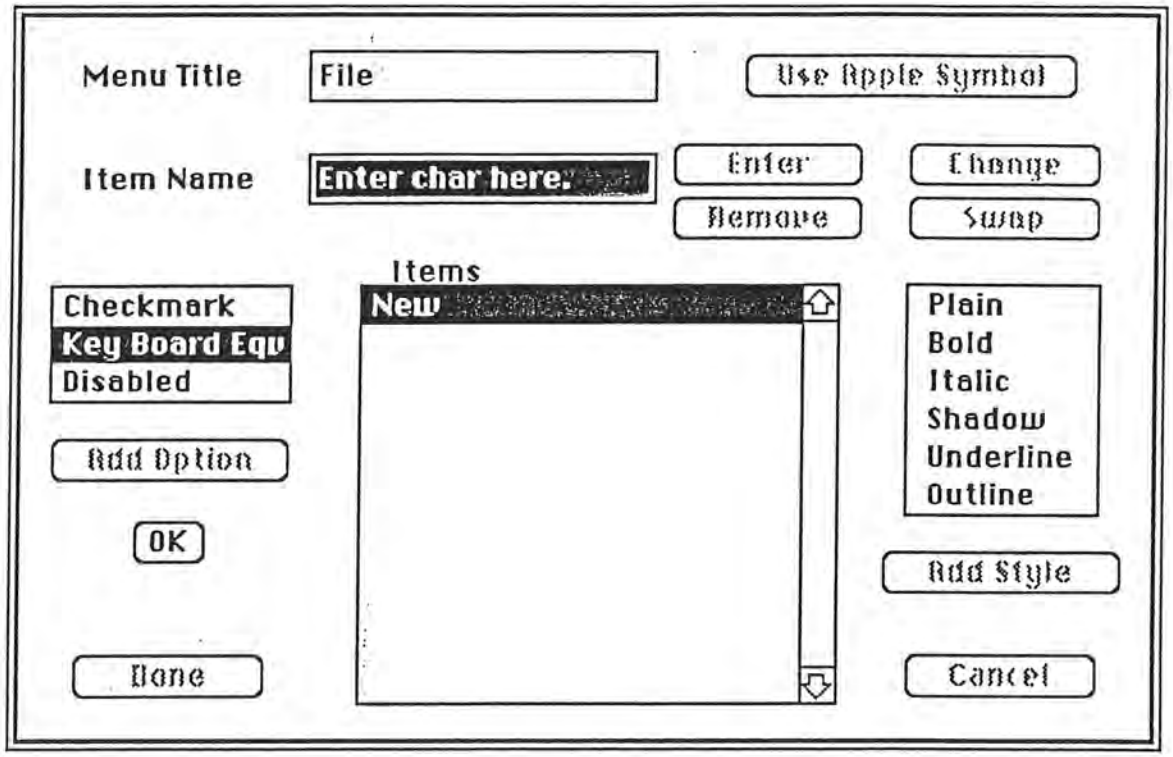



**Figure 19. Original RezDez menu entry dialog.**

b. New menu entry dialog.

In the new menu entry dialog (Figure 20), all of the options and styles are listed as check box items. This not only allows more than one style or option to be added per menu item, but it gives the user the freedom to check any box desired at any time. Also included in this dialog is the ability to attach icons to menu items by its **With Icon** option.

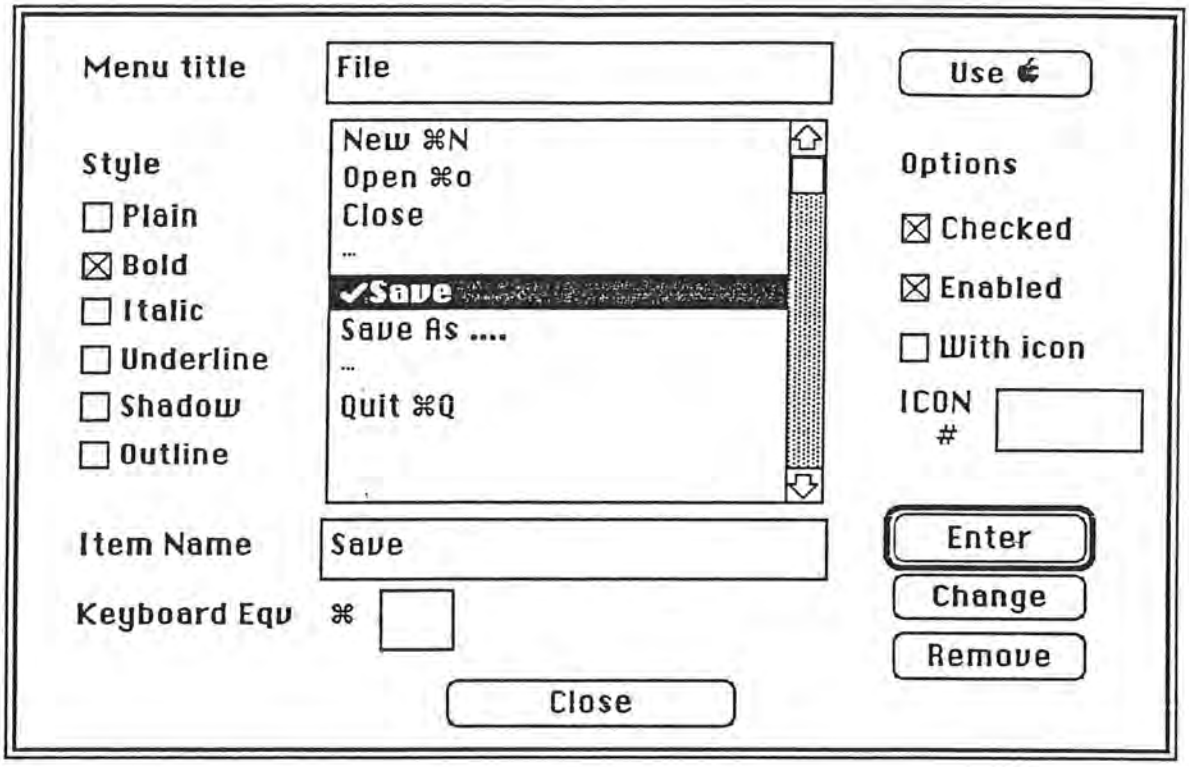



Figure 20 New RezDez Menu Entry Dialog

## 7. Enhancements to Dialogs

One limitation of the original dialog editor was that it allowed only 2 kinds of windows for dialogs, when in fact, Macintosh allows any of the 6 standard kinds of windows to be used for dialogs. Therefore, the following dialog is added to allow the user the choice of any of the 6 standard window types (Figure 21).

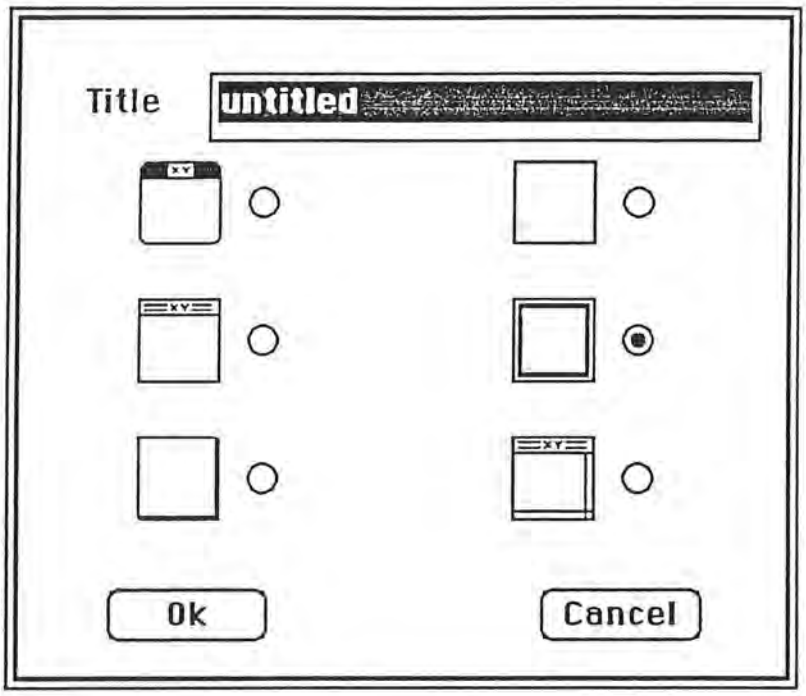



**Figure 21. RezDez Selection Dialog**

Also added to the dialog editor is the ability to include pictures in dialogs. Pictures from the system's clipboard can now be added. Current work is underway to also include pictures from a MacPaint format or a PICT format.

## C. Evaluation and Enhancements to Code Generator

The code generator [Armstrong 88] handles only the user interface portion of the source code. As user-defined procedures and reusable components are added to an application, additional information is required for the code generator to incorporate those routines correctly. In order to do this, a third USES relation file was added in addition to the sequence command file and the control list file that already existed. The USES file is a text file that contains a list of the

user-defined units and reusable component units in the order that they are to appear in the project, i.e. their compiler build order. This information is inserted in the project build order file for the application and in the uses clause of the generated user procedure unit.

```
DLOG_ABOUT
CALCULATOR
DLOG_CALC
```

Figure 22. Sample Uses Relation File

```
ITEMHIT = INIT ;
.;
ITEMHIT = MENUBAR ;
  ITEMHIT = ﷺ [256=0] ;
    ITEMHIT = about calculator [256=1] ;
      DO [DLOG_ABOUT___DO_DLOG_200_DLOG_ABOUT] ;
    .;
  .;
  ITEMHIT = CALCULATOR [257=0] ;
    ITEMHIT = Calculate... [257=1] ;
      DO [DLOG_CALC___DO_DLOG_128_DLOG_CALC] ;
    .;
    ITEMHIT = Quit [257=2] ;
      QUIT ;
    .;
  .;
.;
```

Figure 23. Sample Sequence command file.

The DO clauses in the sequence command file are the user procedures. The format of the parameter in the DO clause is the unit name of the procedure, followed by 3 underscores, followed by the procedure/function name.

```
{List of Units in Build Sequence for Project: calcApp}
{1: Declarations_calcApp}
{2: Globals_calcApp}
{3: SystemCalls_calcApp}
{4.1:DLOG_ABOUT}
{4.2:CALCULATOR}
{4.3:DLOG_CALC}
{4.4: UserProcedures_calcApp}
{5: SimpleAlert_calcApp}
{6: SimpleDialog_calcApp}
{7: no dependant group units in this project}
{8: MenuProc_calcApp}
{9: MenuCase_calcApp}
{10: GoAway_calcApp}
{11: Initialize_calcApp}
{12: Procedures_calcApp}
{13: MainEvent_calcApp}
{14: Main_calcApp}

{The Resource File to use with this project is: Unknown }
```

Figure 24. A sample project build order file.

```
{-------------------------------------------------------}
{     Copyright 1988, Oregon State University.          .}
{                                                       .}
{ This file is the  UNIT for theUser generated procedures created by .}
{the DO verb commands. They are empty stubs at present                 }
{-------------------------------------------------------}
UNIT  UserProcedures_calcApp  ;

INTERFACE
      USES
            Declarations_calcApp, DLOG_ABOUT,CALCULATOR,DLOG_CALC ;

PROCEDURE DLOG_ABOUT___DO_DLOG_200_DLOG_ABOUT{parameters};
PROCEDURE DLOG_CALC___DO_DLOG_128_DLOG_CALC{parameters};

IMPLEMENTATION
{  User  Generated  Procedures/Functions}

PROCEDURE DLOG_ABOUT___DO_DLOG_200_DLOG_ABOUT { };
BEGIN
{USER CODE IS IN ANOTHER UNIT}
      DO_DLOG_200_DLOG_ABOUT
END; { DLOG_ABOUT___DO_DLOG_200_DLOG_ABOUT }


PROCEDURE DLOG_CALC___DO_DLOG_128_DLOG_CALC { };
BEGIN
{USER CODE IS IN ANOTHER UNIT}
      DO_DLOG_128_DLOG_CALC
END; { DLOG_CALC___DO_DLOG_128_DLOG_CALC }

END.{UserProcedures_calcApp  }
```

**Figure 25. A Sample of User procedures unit generated by O.S.U.**

# V. O.S.U.: The solution?

## A. Limitations

Direct manipulation user interface management systems such as O.S.U. largely overcome the problem of use difficulty, but even O.S.U. requires knowledge of the Macintosh software architecture. O.S.U. is for programmers, not end-users. Although aimed at wide-spectrum prototyping, O.S.U. in its current state, is too limited in functionality. O.S.U. cannot, for example, generate itself.

O.S.U is intimately connected to the Macintosh, and would require extensive re-writing to be ported to another system such as X-Windows. It is doubtful that portability is a desirable goal of such systems, but availability should be made a high priority. O.S.U. is available to a limited number of researchers.

## B. Conclusion

Though our preliminary results may be informal and small, we believe it's an indication that O.S.U. can increase the programmer productivity dramatically, as well as improve the quality of applications produced. Integration of CASE with UIMS is certainly another major leap for UIMS. We anticipate that eventually with the merger of software accelerators, we can achieve our goal of a 100 - 1000 fold increase in programmer productivity.

# Appendix

## A. Project Statistics

### 1. O.S.U.

- Application size - 553K
- Lightspeed Pascal Project
  - 109 units
  - over 487K of executable source
  - over 59,000 lines of source code

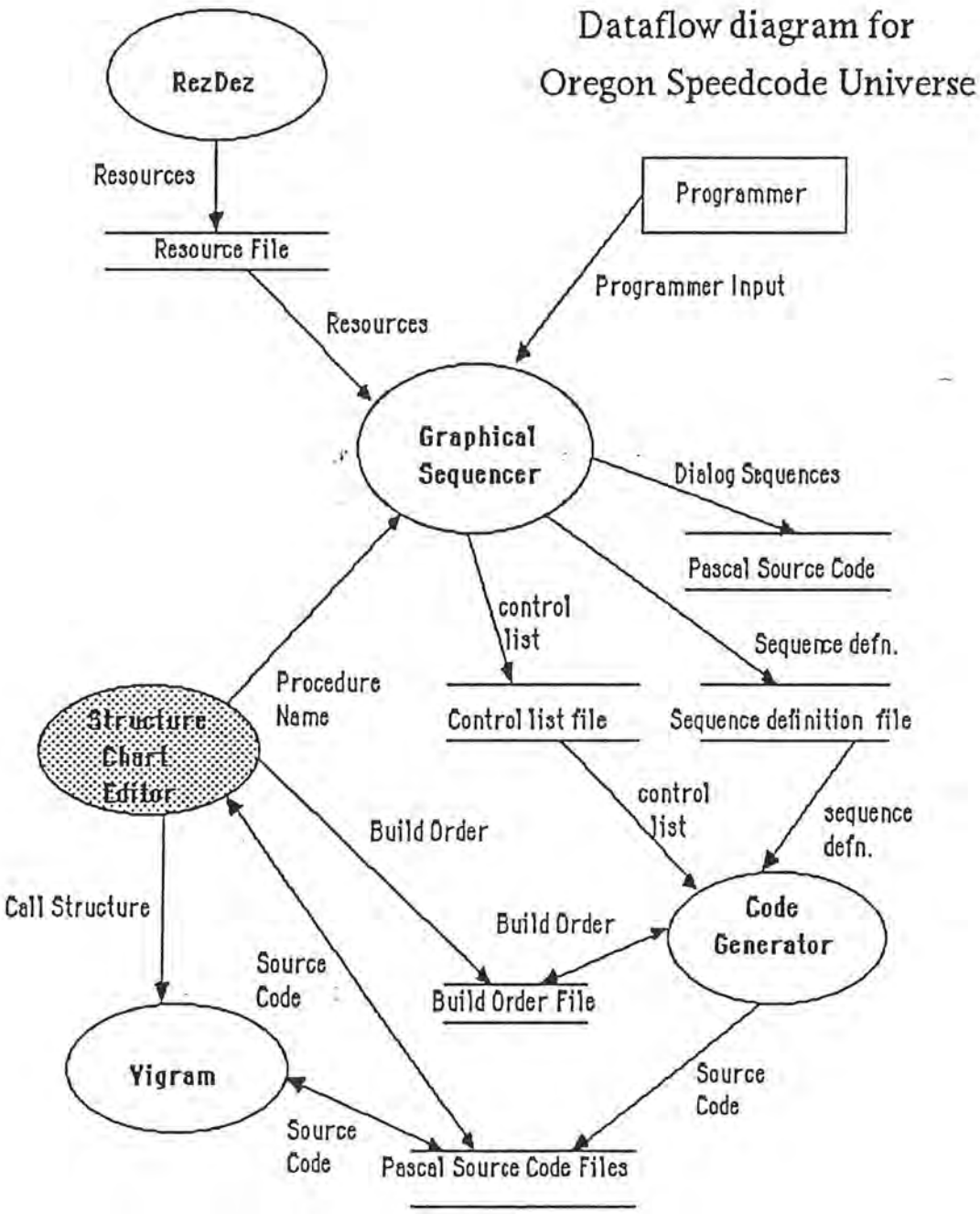
### 2. Structure Chart Editor

- 9 units
- over 27K of executable source
- over 4,000 lines of source code

### 3. Enhancements & Modifications

- 17 units modified
- over 6,000 lines of new code written

## B. Dataflow Diagram of O.S.U.



# References

[Armstrong 88] Armstrong, James, "Code Generation In The Oregon Speedcode Universe", Dept. Of Computer Science **Tech. Report** 88-60-15, Oregon State University, Corvallis, OR.

[Booch 86] Booch, G., "Object-Oriented Development," **IEEE Trans. on Software Engineering**, Vol. SE-12, No.2, Feb. 1986, pp. 211 - 221.

[Bose 88] Bose, S. RezDez, "A Graphical Tool for Designing Resources in OSU", Dept. of Computer Science **Technical Report** 88-60-2, Oregon State University, Corvallis, OR.

[Buxton 83] Buxton, W., Lamb, M. R., Sherman, D., Smith, K. C., "Towards a Comprehensive User Interface Management System," **Computer graphics : SIGGRAPH '83 Conference Proceedings**. Detroit, Mich. Vol. 17, no. 3. July 25-29, 1983. pp. 35-42.

[Cardelli 85] Cardelli, L. and R. Pike, "Squeak: A Language for Communicating with Mice", **Computer Graphics,** July, 1985, pp. 199-204.

[Cardwell 87] Cardwell, Gilbert F.,"A Good Interface Is Difficult To Design," **Computer Graphics,** March 1989, pp. 105-109.

[Chia 89] Chia, Shyang-Wen, **Master paper**, Dept. of Computer Science, Oregon State University, Corvallis, OR., 1989.

[DeMarco 78] DeMarco, T., **Structure Analysis and System Specification**, Yourdon Press, New York 1978.

[Flecchia 87] Flecchia, M.A. and R.D. Bergeron, "Specifying Complex Dialogs in Algae," **Proc. SIGCHI + GI 87**, ACM, New York, 1987, pp. 229-234.

[GIIT 83] Graphical Input Interaction Technique (GIIT) Workshop summary. **Computer Graphics**, vol. 17, no. 1, January, 1983, pp. 5-30.

[Hayes 85] Hayes, P.J., P.A. Szekely, and R.A. Lerner, "Design Alternatives for User Interface Management Systems based on Experience with Cousin," **Proc. SIGCHI 85**, ACM, New Your, 1985, pp. 169-175.

[Henderson 87] Henderson, D. A. Jr., "The Trillium User Interface Design Environment," **Proc. SIGCH'86: Human Factors in Computer Systems**, Toronto, Ont., Canada. April 5-7, 1987, pp. 279-284.

[Hill 86] Hill, R.D., "Supporting Concurrency, Communication, and Synchronization in Human-Computer Interaction: The Sassafras UIMS," **ACM Trans. Graphics**, July, 1986, pp. 179-210.

[Hix 89] Hix, Debroah, "Generations of User Interface Management Systems: Their evolution and relationship to software engineering," submitted to **IEEE Software**, Feb. 1989.

[Hsieh 88] Hsieh, Chia-chi, "A Graphical Editor for Pascal Programming on Macintosh," Dept. Computer Science **Technical Report** 88-60-4, Oregon State University, Corvallis, OR. 97331.

[Hudson 86] Hudson, S.E. and R. King, "A Generator of Direct-Manipulation Office Systems", **ACM Trans. Office Systems**, April, 1986, pp. 132-163.

[Jacob 86] Jacob, R.J.K, "A State-Transition Diagram Language For Visual Programming,", **Computer**, Aug. 1985, pp. 51-59.

[Kasik 82] Kasik, D.J., "A User Interface Management System," **Computer Graphics**, vol 16, no. 3, 1982, pp. 99-106.

[Lewis 89] Lewis, T.G., Handloser III, F.T., Bose,S and S. Yang, "Prototypes From Standard User Interface Management Systems," **Computer**, vol. 22, no. 5, May, 1989, pp. 51-60.

[McClure 89] McClure, Carma, **CASE Is Software Automation,** Prentice Hall, New Jersey, 1989.

[Musa 85] Musa, J. D., Software Engineering: The Future of a Profession, **IEEE Software**, vol. 2, no.1, January 85, pp. 55-62.

[Myers 87] Myers, B.A., "Creating User Interaction Techniques by Demonstration," **IEEE Computer Graphics and Applications**, vol. 7 no. 9, Sept 87, pp. 51-60.

[Myers 89] Myers, B.A., "User Interface Tools: introduction and survey", **IEEE Software**, vol 6, no. 1, Jan. 1989, pp. 15-23.

[Olsen 83] Olsen, R.D. Jr., and E.P. Dempsey, "Syngraph: A Graphical User Interface Generator," **Computer Graphics**, July, 1983, pp. 43-50.

[Olsen 85] Olsen, R.D. Jr., E.P. Dempsey, and R. Rogge, "Input-output Linkage in a User Interface Management System," **Computer Graphics**, July, 1985, pp. 225-234.

[Pfaff 85] Pfaff, G., ed. **User Interface Management Systems**, Springer-Verlag, Berlin, 1985.

[Prototyper 87] SmetherBarnes Prototyper User's Manual. P.O.Box 639, Portland, Or. 97207.

[Raghu 89] Raghu, J., "Adding Functionality to a Vacuous Prototype", Dept. of Computer Science **Tech. Report** 89-60-4, Oregon State University, Corvallis, OR. 97331.

[Schmucker 86] Schmucker, K.J.,"MacApp: An Application Framework,", **Byte**, Aug. 1986, pp. 189-193.

[Sibert 86] Sibert, J.L., W.D. Hurley and T.W. Bleser, "An Object-oriented User Interface Management System", **Computer Graphics**, Aug. 1986, pp. 259-263.

[Yang 89], Sherry Yang, T. G. Lewis, & C. Hsieh. "Integrating Computer-Aided Software Engineering and User Interface Management Systems", **Proceedings of the 22nd Annual Hawaiian International Conference on System Sciences**, Vol. II, 1989.