1988

Generating Multiple User Interfaces for Multiple Application Domains

Mahesh Hassomal Dodani
University of Central Florida

GENERATING MULTIPLE USER INTERFACES FOR MULTIPLE APPLICATION DOMAINS

by

Mahesh Hassomal Dodani

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
the Department of Computer Science at
the University of Central Florida
Orlando, Florida

December 1988

Major Professor: David A. Workman

ABSTRACT

This Ph.D. dissertation presents a classification scheme for User Interface Development Environments (UIDEs) based on the multiplicity of user interfaces and application domains that can be supported. The SISD, SIMD and MISD **[S= Single, I= user Interface(s), M= Multiple, D= application Domain(s)]** generator classes encompass most of the UIDEs described in the literature. A major goal of this research is to allow any user to develop a personalized interface for any interactive application, that is, the development of an **MIMD UIDE.**

Fundamental to the development of such a UIDE is the complete separation of the user interface component from the application component. This separation necessitates devising less tightly coupled models of the application and user interface than have been reported to date. The main features of the MIMD UIDE model are as follows.

[1] Interactive applications are modeled as editors providing a set of functions that manipulate 2-dimensional graphical objects.

[2] **Interactive data structures** are introduced for maintaining and manipulating both the internal and external representation(s) of application information as a single unit. These external representations form the basis for presenting internal information to the user.

[3] Since interaction with the user must be the sole responsibility of the user interface component, function interaction is modeled as follows. Application functions are modeled as a **set of services.** Each service processes a (set of) parameter(s) independently. For each service in the application, a corresponding **service interface object** is defined in the user interface component. The service interface object interacts with the user to specify the required (set of) parameter(s), calls the associated service within the application, and displays the result of the service to the user.

Using the above model, the user interface component is modeled to allow personalized specifications at all levels; including the internal entities of the interactive system, the characteristics of the display of information, and the interaction tasks, techniques and devices used for parameter specification.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Since the dawn of computing, scientists have directed their research at the core of computer systems, in such topics as hardware and software. Research interest diminishes as we move outward from the core of the system to its boundary. Historically, the **terminal** was considered as the boundary of the system. However, it is now widely recognized that the user forms the critical component of the computer system and determines whether the system as a whole - the man-machine system - **works** or not.

The evolution of computer science has followed a **system-oriented** trend. That is, scientists have always been concerned with making the system work, considering user-specific issues only when research on system issues has been completed. This system-oriented approach is typified by the history of editing systems in support of programming. Until recently, program editing was supported by text editors, which consider programs as text or a sequence of characters. Text editors themselves have evolved from batch-oriented to interactive, from line-oriented to screen-oriented, and most importantly from providing no on-line assistance to providing sophisticated help features. However, from the programmer's viewpoint, text editors are a limited programming tool - they provide no support for the programming process itself. In fact, in conventional systems, the compiler is the first productive tool in support of the programming process - the text editor is merely a tool for creating the input for the compiler. Once the technology of compilers was fully developed, concerns for the programming process shifted to the programmer. This concern resulted in **syntax-directed editors,** a true program editor. A syntax-directed editor has knowledge of the programming language, and editing is done in terms of program constructs. Thus, a syntax-directed editor can always ensure syntactic correctness and is capable of providing programming support to the user. Technology in syntax-directed editing has reached a point where facilities are provided in support of all aspects of programming (including semantic analysis, execution and debugging) as typified by the Cornell Program Synthesizer [1].

The design of such **interactive systems** does not, however, consider user behavior. The approach has not been one of first understanding the user's model and then designing systems to suit this model. To take the editing example further, the design of a program editor was not carried out after a careful evaluation of the user's model of programming. Rather, an ad-hoc approach is employed - it would obviously be more productive if the editing and compilation phases were merged. This ad-hoc approach to the design of the user aspects can be attributed to the attitudes of the **technologist** and the **designer.**

The attitude of the technologist is that the user is made more productive by massive technological advances, such as bigger and faster machines. As an example, the advent of high resolution graphical devices have improved the interface for the user as opposed to the teletypewriter interface. However, as is currently being discovered, new technologies introduce their own problems. Currently there exists a new set of display problems with graphical devices that did not exist with teletypewriters, e.g., what information should be displayed, how should it be displayed, how much should be displayed, how do we ensure that the user's attention is drawn to specific information, etc.

The attitude of the systems designer is that, being human, he/she can rely on common sense and intuition to predict what will be easy for the user. However, the problems with this ad-hoc approach are twofold: designers cannot evaluate their intuition and the designer's intuition do not necessarily match the user's.

Moran [2] describes the resulting state of affairs in harsh terms:

> **The computer systems commonly available today are indeed wonderful. ..... Yet, except for occasional examples of inspired design, users find most of the systems they use to be arcane, idiosyncratic, inconsistent, and seemingly more difficult to learn and to use than necessary.**

A common sight throughout the computer science community is that of a frustrated user pouring over a thick manual or in deep consultation with the local guru.

In the last decade, the emphasis in computer systems has shifted from a systems-oriented view to a **user-oriented view.** The user is now accepted as being the most crucial component of the system and a large percentage of system design is based around the user. This dramatic shift in emphasis can be attributed to the following reasons.

[1] Computer systems are ultimately employed by computer-naive users. To be productive, naive users require an effective user interface. Initially, two major application areas brought the problems of designing effective user interfaces to the limelight. In database management systems, the primary concerns were in providing an effective dialogue interface. The primary goal was to provide a dialogue system in which a user could converse **naturally** with the system. On the other hand, data-entry systems require an interface that minimizes errors and guides the user for the entry of data through prompts and menus. More recently, and most importantly, personal computer users demand an effective user interface in general, covering every aspect of the computer system.

[2] The tremendous advances made in the field of computer graphics in both hardware and software have provided the necessary technology for the production of sophisticated user interfaces. Current graphics technology is characterized by the following:

[a] a wide range of input/output devices, e.g., joysticks, light pens, mouse, etc. as input devices; and a variety of high resolution graphical monitors as output devices;

[b] support for a range of graphical attributes, e.g., color, intensity, highlighting, etc.;

[c] routines for creating graphical images, e.g., lines, circles, boxes, etc.;

[d] fast and efficient algorithms for graphics display; and

[e] the ability to subdivide a screen into logical windows (or virtual output devices) and the associated screen management routines.

[3] The acceptance that interactive computer systems are more productive and effective for the user. Interactive systems are typified by

[a] the screen always displays the current state of the information that the user is manipulating, e.g., the files available on a particular disk, a document being edited, the relations in a database, etc.;

[b] the screen can also display the options available in the current context to the user, e.g., in the form of menus;

[c] the screen is divided into logical areas or windows, each containing related information, e.g., an editing window; and

[d] user interaction is typically as follows - the user selects a function from a menu and identifies parameters if necessary; the system processes the function and updates the information on the screen according to the result of the function.

Current research is therefore concerned with graphical interactive user interfaces. The research has been mainly concerned with understanding the nature of interactive systems as a whole with the aim of aiding the interface designer. To depict the diversity of the area, the following (reproduced from [3]) enumerates the keywords and phrases which describe the areas of concern:

[a] **Human Factors:** comfort of the workstation, orientation of the display, continuity of conversation, correct pacing, user vs. machine control of flow, understanding the user's model of the process, semantics in terms of the user's domain, provision of feedback at appropriate times and in the user's terms, benefits of modeless commands, provision of friendly and forgiving systems, ability to recover from mistakes, confirmation of dangerous actions, choice of expression, extensible command languages, use of hierarchical languages, use of novice and expert modes for prompts;

[b] **Devices:** terminal/satellite/stand-alone configurations of processors, buttons, tablets, light pens, joysticks, shaft encoders, virtual devices, locators, valuators; and

[c] **Techniques:** use of menus, windows, dragging, motion, depth cuing, stereo pairs, color, feedback of input actions.

In the remainder of this introduction we attempt to clarify how human factors, devices and techniques relate in the design of modern interactive systems and in particular, the design of their user interfaces. The next section describes graphical interactive systems with the aim of developing appropriate models and methodologies for their development. The following section concentrates on the user interface component of graphical interactive systems. In particular, it focuses on the issues of developing user interfaces. A simple classification framework is developed in this section to differentiate between the various approaches to developing user interfaces. The final sections describe the goals of this research and the organization of this dissertation.

1.1 Graphical Interactive Software Systems

The advent of sophisticated graphical input and output devices has led to a proliferation of interactive graphical software systems as the medium of communication with a digital computer. The principal problem underlying the research in interactive systems is how can **better** interactive systems be built. The aim is to identify concrete guidelines (preferably as a formalism and its associated methodologies) that can aid the designer in building **better** interactive systems. The term **better** is interpreted by researchers in different ways, but the principal implication is better human factors. The above problem is in the domain of **software engineering,** the study of methodologies for building software systems. To date, software engineering has also adopted the system-oriented perspective. The formulation of the software life cycle has been one of its primary contributions. The software life cycle identifies individual manageable phases of the software, including but not restricted to requirements specification and analysis, high-level design, coding, testing and debugging, and maintenance. These phases are tied together through management and communication aspects, including documentation, budgeting, personnel deployment, project review, scheduling and configuration management. Software engineering research has concentrated on developing software development methodologies to support the life cycle, and the implementation of automated systems to support the development methodologies.

Researchers are currently trying to shift the emphasis of software engineering to a user-oriented perspective. In this perspective the additional requirements and constraints imposed by highly interactive systems are considered. The main goal of the research is to develop suitable user-oriented technologies for effective and efficient interactive software engineering. Jurg Nievergelt [4] identifies the following three phases of any emerging technology for system development.

[1] **The Ad-hoc Phase:** In this phase, researchers use ad-hoc approaches to build effective and efficient systems. The main emphasis of this phase is to understand the complexity of the system with the aim of identifying their desirable characteristics as well as defining their properties.

[2] **The Model Phase:** This phase concerns itself with the development of suitable models of the system which form the basis for methodologies for building effective and efficient systems. We can identify two radically different approaches to system modeling. The **formal** approach applies existing formalisms and theories to develop models and their associated methodologies. On the opposite end of the spectrum of possible approaches, the **practical** approach relies on the best engineering principles and techniques that have been proven in practice to develop suitable models and their associated methodologies.

[3] **The Theory Phase:** The final phase is concerned with identifying ideal models and developing a solid theoretical foundation for these models and their associated methodologies. The main focus of this theoretical foundation is to provide the necessary base for

[a] proving the correctness of the model and its methodologies,

[b] developing suitable formalisms to define system properties,

[c] developing suitable formal frameworks for system development methodologies, and

[d] determining the complexity of the model, its methodologies and the system.

The model and theory phases are dependent on each other as formalisms necessitate ideal models and models can only be ideal if they are supported by a solid theoretical foundation. The technology becomes **mature** when there exist abstract general models that are supported by effective and efficient formal development methodologies.

The first step towards a mature technology for developing interactive systems is the definition of interactive systems with the aim of identifying their properties. A survey of the vast literature on interactive systems shows that no widely accepted definitions exist. One of the problems is that graphical interactive systems are complex creatures and their complexity is not very well understood yet. To understand the complexity of graphical systems, consider the Dynabook: Alan Kay's dream of a **dynamic medium for creative thought** [5]. The Dynabook can be considered as a technically sound vision of the ultimate potentialities of interactive computer graphics:

> **Imagine having your own self-contained knowledge manipulator in a portable package the size and shape of an ordinary notebook. Suppose it had enough power to outrace your senses of sight and hearing, enough capacity to store for later retrieval thousands of page-equivalents of reference materials, poems, letters, recipes, records, drawings, animations, musical scores, waveforms, dynamic simulations, and anything else you would like to remember and change.**
>
> **We envision a device as small and portable as possible which could both take in and give out information in quantities approaching that of human sensory systems. Visual output should be, at the least, of higher quality than what can be obtained from newsprint. Audio output should adhere to similar high-fidelity standards.**
>
> **There should be no discernible pause between cause and effect. One of the metaphors we used when designing such a system was that of a musical instrument, such as a flute, which is owned by its user and responds instantly and consistently to the owner's wishes. Imagine the absurdity of a one-second delay between blowing a note and hearing it!**

We can look at our own world which is filled with interactive systems and interactive communication for some metaphors that illuminate the desirable properties of interactive graphical systems. Some of the more useful metaphors are described below.

[1] **Instruments:** As is noted by Kay, good instruments provide **immediate feedback** to the user. They must be crafted with **care** and **precision** and must be **redesigned** as components and techniques evolve and change. Consider the relationship of the harpsichord to the clavicord to the piano; a progression which is characterized by changes in the striking or plucking mechanism.

[2] **Tools:** Kay has also noted the importance of **power, flexibility** and **generality** in tools. Tools should also be as **adaptable** as possible; consider the Swiss Army knife. They must also be **robust,** i.e. be able to handle user errors.

[3] **Vehicles:** Vehicles must **respond sensitively** and **repeatably;** consider steering an automobile and landing a plane. They must be engineered to provide **adequate feedback;** consider the design of the airplane cockpit display. Most importantly, vehicles must be **suitable** for the purpose; consider bicycling up Mt. Everest.

[4] **Games:** Computer and video games provide examples of some of the best interactive graphical systems available today. Games must continuously **captivate the interest** of the players. Therefore they must be **engrossing** and **fun.** Games must never allow the player to lose confidence or patience; thus they must be based on **rules** or **structure.** Finally, games must **challenge** the player, thereby encouraging **mastery** and **growth.**

[5] **Human-to-human dialogues:** Many researchers (e.g., [6]) have attempted to define the characteristics of user-computer dialogues through a careful examination of human-human conversation. The attributes of human-human conversations that need to be emulated are

[a] human responses are **appropriate** and **sensitive** to the current context of the conversation and are based on intelligent processing and interpretation of the other person's messages and behavior, and

[b] human dialogue is **multi-dimensional** and **multi-level** allowing the communication to proceed along a number of paths and a conversational context to be started and re-

started at will.

Thus, as summarized in [7]

> **Interactive graphics systems can be instruments for expression or measurement, tools to cause effects, vehicles for exploration, games for challenge or amusement, and media for communication.**

In the search for a more formal definition and models of interactive systems, a workshop on the methodology of interaction was set up in 1979 [8]. This workshop comprised the principal researchers in the area of interactive graphical systems. As far as defining interactive systems, the workshop indicates that

> **The search for a formal definition of the term 'interactive system' was given up at a very early stage.**

However, a list of desirable properties of an interactive system were enumerated, and is reproduced below.

[1] There must be a priori general understanding of the application domain common to both man and machine.

[2] There must be a closure in the common understanding of the current context and goal.

[3] This closure must be maintained as the discourse proceeds.

[4] The responsibility for achieving goals must be shared between man and machine.

[5] Initiative should be balanced between man and machine.

[6] The system should be adaptive in a variety of senses to the user, the current context, the available input/output devices and communications, and the application domain.

[7] The system must have the capability to explain its assertions.

[8] The system should have the ability to transfer gracefully between query, explanation and knowledge acquisition modes.

The next step in the process of developing a mature technology for interactive systems is the search for ideal models. The important characteristics of an ideal model for interactive software systems are the following.

[1] The model must be user-oriented, that is, it must allow end users to be intimately involved with the process of interactive software engineering.

[2] The model must be general enough to describe any interactive software system.

[3] The model must allow for efficient and effective development methodologies for interactive systems.

Many approaches to the modeling of interactive systems have been proposed. The aim of an interaction model is to be able to develop guidelines for a design methodology of interactive systems. We have identified two approaches to system modeling; the formal and practical approaches. The next two subsections discuss these approaches to interactive system modeling.

1.1.1. Formal Interactive System Models

The formal approach applies known formalisms and theories to model interactive systems. Specifically, it draws on the formalisms and methodologies of conventional system-oriented software engineering as the basis for developing interactive systems. This formal approach is characterized by the following

[a] a division of the development process into a number of independent phases,

[b] the use of formal notations as specification mechanisms within each phase, and

[c] the use of formal verification procedures to validate the development of the system between phases.

Some of the major formal models are surveyed briefly below.

[1] **NETWORK MODELS.** The network models treat user and system as equivalent elements in the overall process. The individual tasks performed by both the user and the system are described in terms of expected performance and logical relationships. The relationships define a network of tasks which is used as a performance model of the user-computer system. Thus, network models allow performance data about user and computer system to be integrated in a single model. However, performance data must be provided for each task, as must rules for combining performance data from individual tasks to obtain aggregated performance predictions. This is often difficult because of questionable or lacking empirical data, and because interactions between tasks (especially cognitive tasks or tasks performed in parallel) may be very complex. Therefore, network models are usually used to predict either the probability of failure or success, or the completion time of an aggregated set of tasks.

[2] **CONTROL-THEORY MODELS.** These models are based on control theory, statistical estimation and decision theory; and are usually used to predict overall performance of the user-computer system in continuous control and monitoring tasks. In a dialogue system, this is a model of the control feedback between man and machine. The main factors to be considered are task difficulties, the constraints and facilities of the environment, the disposition and interest of the operator in the task being executed, the response time of the system and the way information is presented [9]. Figure 1.1 illustrates the relation of these factors. Control theory is used to describe the human response to external stimuli using models based on open-loop, closed-loop structures or a combination of these. As an example, compensatory control is a closed-loop structure as illustrated in Figure 1.2. Control-theoretic models are more quantitative than other performance models. They may address user-computer communication broadly, but they ordinarily do not deal with details of the interface, such as display design. Therefore, their utility as an aid to the interactive system designer may be limited.

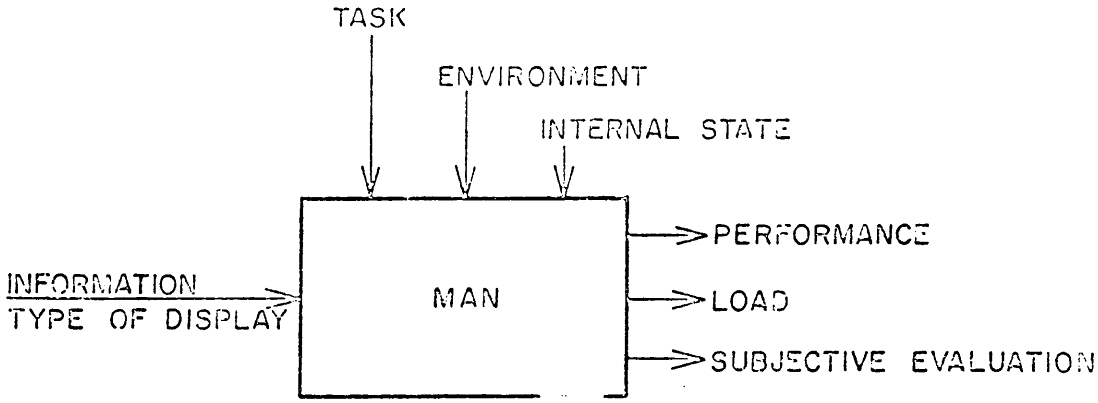


Figure 1.1: The Control Theory Model of Interaction

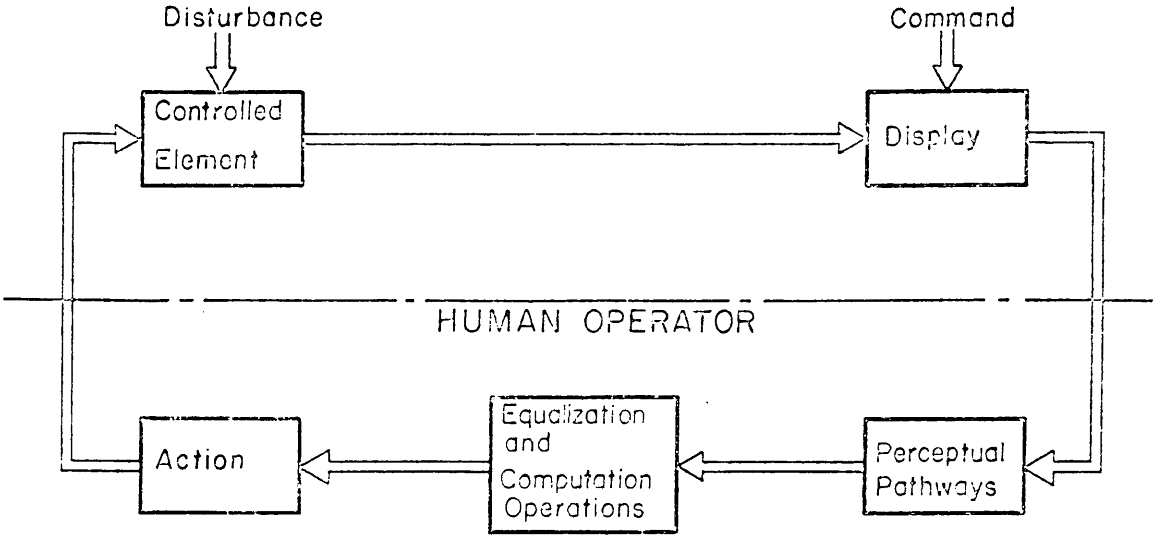


Figure 1.2: An Example of Compensatory Control

[3] **HUMAN INFORMATION PROCESS MODELS.** In general, these models involve a characterization of

[a] the task environment, including the problem and solutions available;

[b] the problem space employed by the user to represent a problem and its evolving solution; and

[c] the procedures developed to achieve a solution.

The method used to develop such models involves intensive analysis of the problem to be solved and of protocols obtained from problem solvers during solution. As a particular example, the **knowledge model** [10] describes a system to support the user's problem solving task. The system includes a knowledge base, a database and a procedure base as basic information and a set of high-level procedures that manipulate these information bases. The procedures include algorithms to translate the external form of information to a suitable internal form, and to generate programs. The basic configuration of the system is illustrated in Figure 1.3.

[4] **COGNITIVE MODEL.** The cognitive model [11] is a concept to organize flexible man-computer dialogues. This model achieves the interactive system design goals of **adequate usability** and **ease of learning** by applying two major principles

[a] decomposition of user tasks as a tree structure, and

[b] the separation of planning a task and performing it.

The cognitive executive system is therefore divided into two parts, the superordinate system which is concerned with making a plan and a subordinate system which automatically performs the plan. This distinction between knowing and doing is incorporated into the dialogue model which allows for three kinds of tours: the **direct way** (i.e., the user's task is immediately performed), **deviation** (i.e., the user requests information and then his task is performed) and **excursion** (i.e., the user learns more about the system information). The

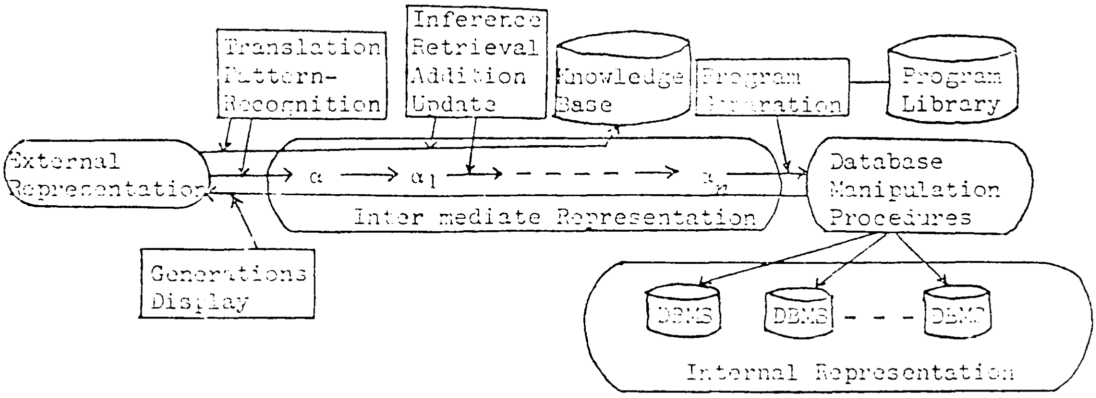


Figure 1.3: The Knowledge Model of Interaction

dialogue model is illustrated in Figure 1.4.

[5] **LANGUAGE MODEL.** Interaction with a computer is typically described as a conversation between man and machine, implying that a language is used in the conversation. The language model [12, 13, 14] is therefore aimed at describing the structure of the conversational language with the aim of understanding the interaction itself. This language consists of two components as depicted in Figure 1.5: the **input language LI** and the **output language LO.** These languages are closely interrelated by a single conceptual model of the processes being performed by the system. The user communicates to the computer through the input language, which operates on the conceptual model. The machine communicates to the user with the output language, which depicts the state of the conceptual model. Each language in turn has an associated **semantic, syntactic** and **lexical** component, all integrated together by the conceptual model of the system. Semantics is the set of meanings associated with the atomic units (i.e., words) and combination of units (i.e., sentences) of the language. The syntactic rules define the grammar that describes how words are combined into sentences, while the lexical rules determine how the system's basic primitives are combined into words. In the output language, the semantics is the displayed image, the syntax determines the organization of the image, and the lexical rules determine the details of the image's appearance. For the input language, the semantics are the commands, the syntax are the words that form the commands and the lexical rules are the interaction techniques and devices that allow specification of the words. There is also an implied communication protocol defining the interconnection and sequencing of the user to computer and computer to user conversations at the lexical, syntactic and semantic levels. For example, lexical inputs are typically echoed; syntactic inputs lead to prompts and semantic inputs cause the displayed image to change.

A synthesis of the most important features and capabilities of all the above models into one single comprehensive model was proposed at the methodology of interaction workshop [8].

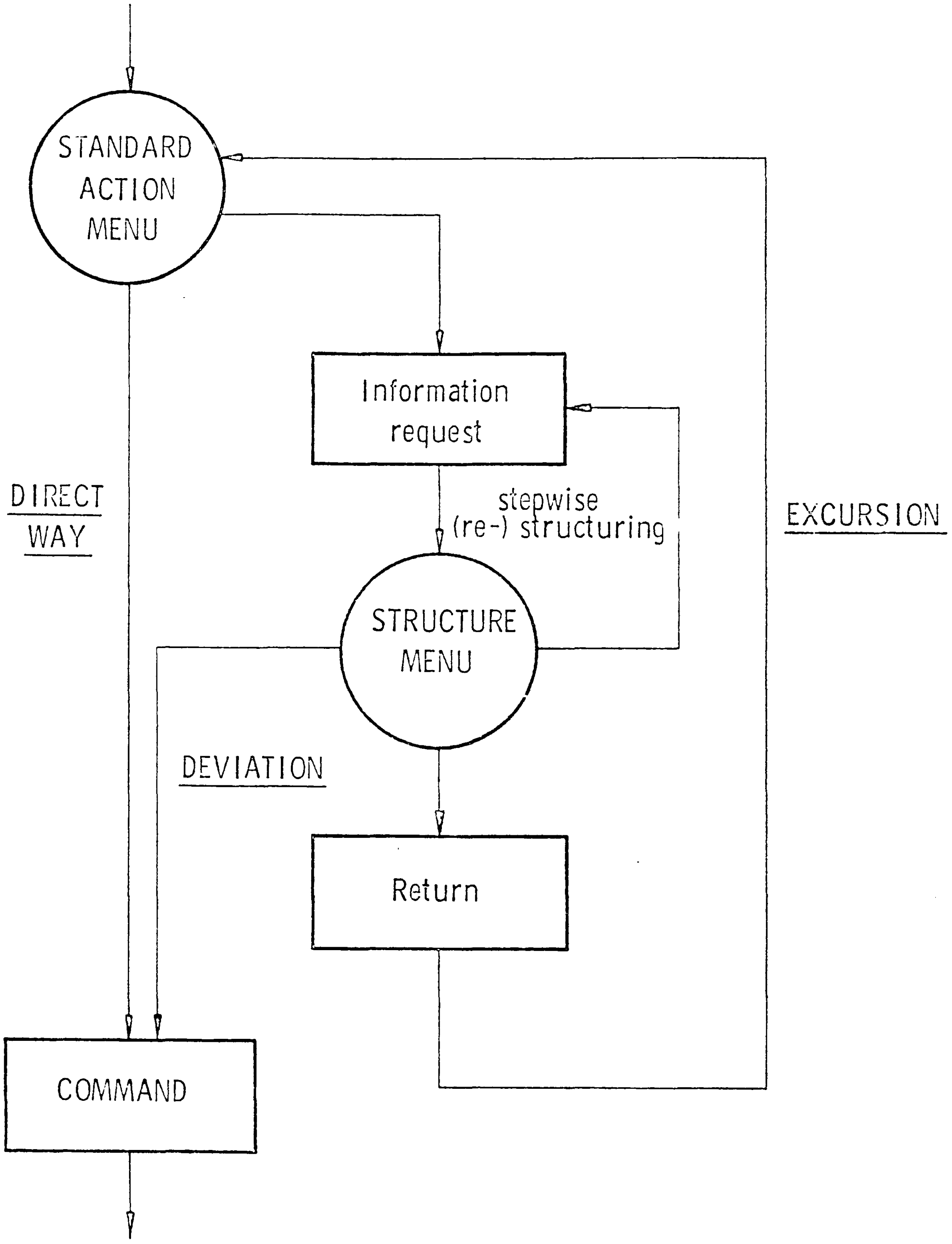


Figure 1.4: The Cognitive Model of Interaction

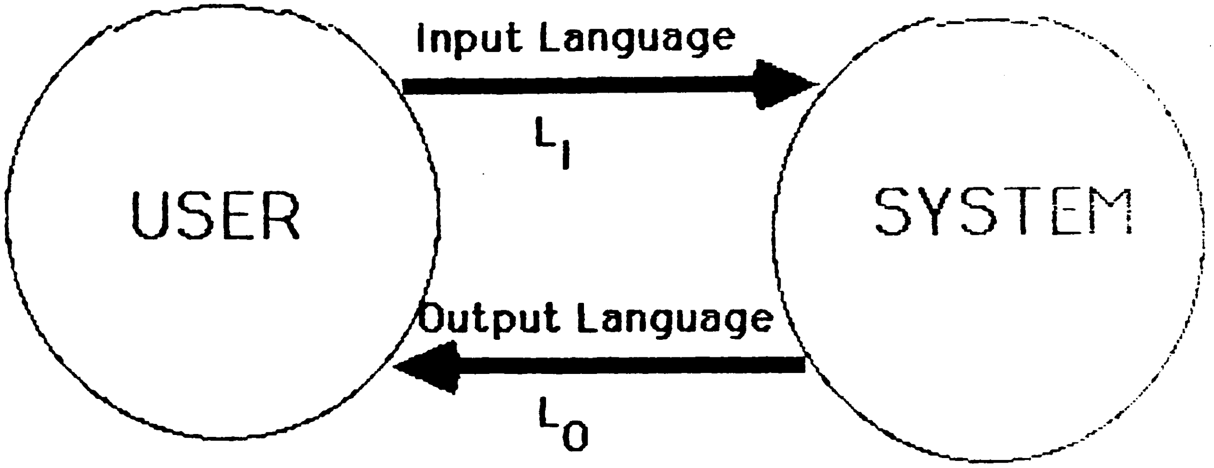


Figure 1.5: The Language Model of Interaction

The model that was formulated was largely based on the language model. The input and output languages were elaborated on further as depicted in Figure 1.6 to arrive at a basic interaction model as illustrated in Figure 1.7. The interaction model was described from the viewpoint of the interaction process as follows.

[1] A user interacting with the system (say via a display) inputs a command(s).

[2] The command is processed lexically, syntactically and semantically and then appropriately routed to the inference processor, task performer or the filter.

[3] With input from the command processor and using the structure of the task, the inference processor essentially builds up the knowledge bases.

[4] The task performer analyzes the input commands, decides on the structure and the task to be performed, updates the application data base, supplies information for inference purposes and also for the filter process which does the feedback generation.

[5] The filter essentially provides the feedback in the context of the command, application database, the user's knowledge database and the task to be performed.

[6] The knowledge database models the user's behavior, is maintained by the inference processor and feeds into the feedback mechanisms.

[7] The application database is updated when tasks are performed.

[8] The feedback generation again goes through semantic, syntactic and lexical phases.

To summarize, the interaction model consists of the following four components.

[1] A **user's model,** i.e., the conceptual model of the information the user manipulates and of the processes applied to this information.

[2] A **command language** with which the user expresses commands.

[3] **Feedback** provided by the system in response to user actions.

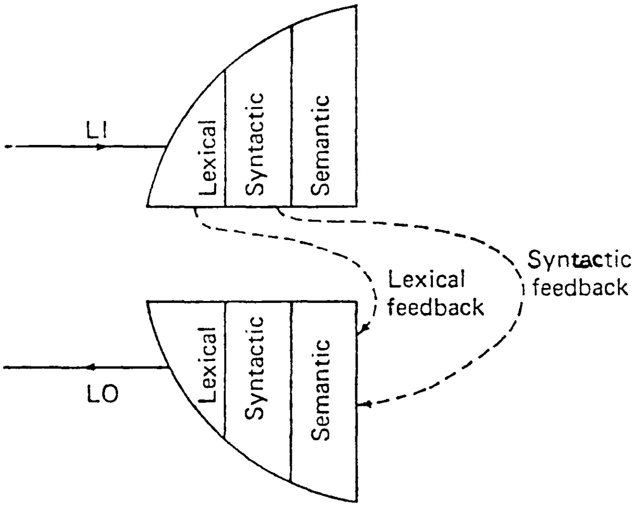


Figure 1.6: The Input and Output Languages

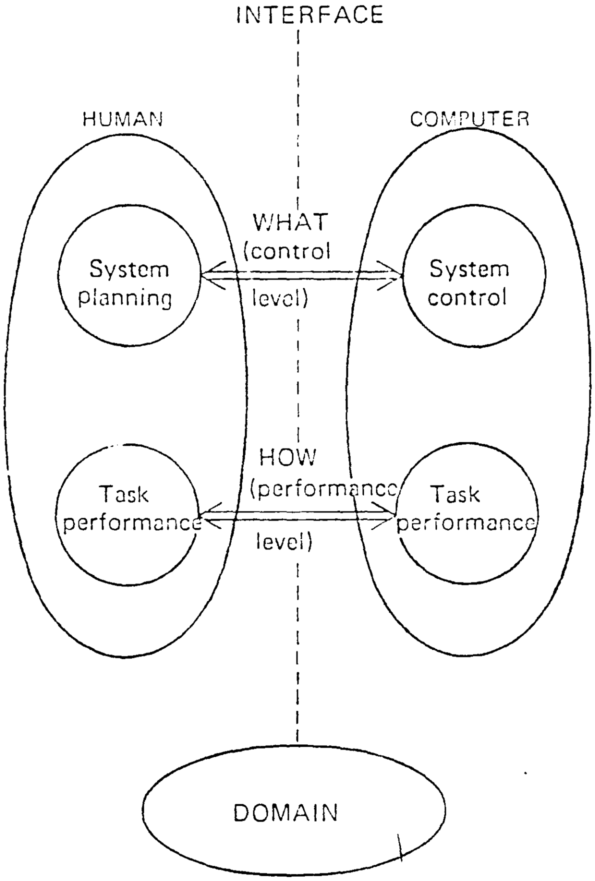


Figure 1.7: The Basic Interaction Model

[4] **Information display** which shows the user the state of the information that is being manipulated.

Using the above model, the design of an interactive graphics system must cover every aspect of the user-computer interface. This ranges from the concepts (i.e., the conceptual model) that the user must deal with to the finer details of screen formats, interaction techniques and device characteristics (i.e., the lexical design of the input language). Most researchers (e.g., [13, 15, 16, 17, 18, 19]) have adopted a top-down approach to the design of interactive systems. As with the software life cycle, the first phase of the design is concerned with **requirements definition** with the aim of understanding the application area and prospective users. Since the approach is user-oriented, this phase is mainly concerned in defining the **user's requirements.** As is suggested in [20],

> **Observing what man does normally during his creative efforts can provide a starting point for the ... designer. In particular, a mathematician does not manipulate equations at a typewriter, nor does a circuit designer prefer a keypunch.**

The advice given in [21] is even more direct:

> **Know the User. Watch him, study him, interact with him, learn to understand how he thinks, and why he does what he does.**

The requirements definition process results in the following

[a] A set of **functional requirements,** or capabilities, which are to be made available by the interactive system. Other design objectives such as learning time, speed of use, error rates, user satisfaction and market appeal are also necessary.

[b] **Design constraints** imposed by the existing equipment, compatibility with other computer systems, implementation time and available resources.

[c] Identifying the **types of users** for which the system is to be designed. The main concern here is defining user characteristics such as personality (i.e., secure/insecure, bold/timid, adaptable/rigid), knowledge (both in skill (novice, intermediate or expert) and in intelligence (low, average or high)), work environment (i.e., defining the levels of stress and motivation)

and the adaptability to change.

The results of the requirements definition phase is used as the basis for defining the input and output languages as defined by the interaction model. For example, defining the input language is also a top-down process, starting with the conceptual model, then the command structure, the syntax and finally the binding of physical devices and interaction techniques. Thus the ideal process of designing interactive systems is described as a top-down process consisting of five phases: requirements definition, conceptual model design, semantic design, syntactic design and lexical design. Each phase is dependent on the previous phase. Implicit in the process is the design of the dialogue between man and machine, as is verification, testing and debugging across all phases.

Green [22] suggests three basic goals of an effective formal design methodology for interactive systems.

[1] A **formal notation** for describing the user interface should be provided by the design methodology. A formal notation is important for two major reasons. It ensures that a description of a user interface is interpreted in a uniform manner by the users of the notation, i.e., the personnel involved in the development of the user interface. More importantly, a formal notation serves as the basis for the implementation of the user interface from its description.

[2] The methodology should provide mechanisms for validating the man-machine interface design. Here we are interested in determining the correctness of the design before its implementation. The experiences of the research on software engineering has shown that it is hard, if not impossible, to show that a design is error free. The major concern is therefore the elimination of major bugs from the design before its implementation.

[3] The methodology should provide an effective framework for evaluating the quality of the interactive system, that is, its effectiveness. However, very little is known about the factors that affect the quality of the interactive system. The methodology should at least provide a

framework for measuring the human factors principles in the design.

We present below two impressive formal design methodologies.

The design methodology proposed by Green [22] is divided into two components, a formal description of the user's model and a formal specification of the user interface based on the user's model. The design methodology builds a separate user's model for every type of user that will be involved.

The description of the user's model comprises two components:

[1] a **task model,** which formally describes the user's view of the problem in terms of atomic tasks. The task model is obtained through an informal task analysis, i.e., the decomposition of the problem into a number of tasks which are atomic from the user's point of view.

[2] a **control model,** which describes the actions that the user can perform.

Thus, the task model describes the tasks to be performed and the control model describes the commands that are provided to perform them, both from the user's viewpoint. Note that the control model must be consistent with the task model.

The notation employed for describing the task and control models is comprised of three components, object definitions, operator definitions, and invariants. Object definitions declare the properties (attributes) of the objects, similar to data structure definitions. Operator definitions describe the conditions for the application of an operator and the effects on the objects. Invariants describe the relations between the operators and objects in the model, and may vary greatly from user to user.

As an example, consider the design of a user interface based on the screen layout depicted in Figure 1.8. The screen is divided into a menu area which depicts the geometrical shapes available to the user, and a work area, where the user creates pictures by arbitrary placement of the available geometric shapes. Three commands are supported: the **place** command allows the user to use a digitizing tablet to select a geometric shape from the menu and

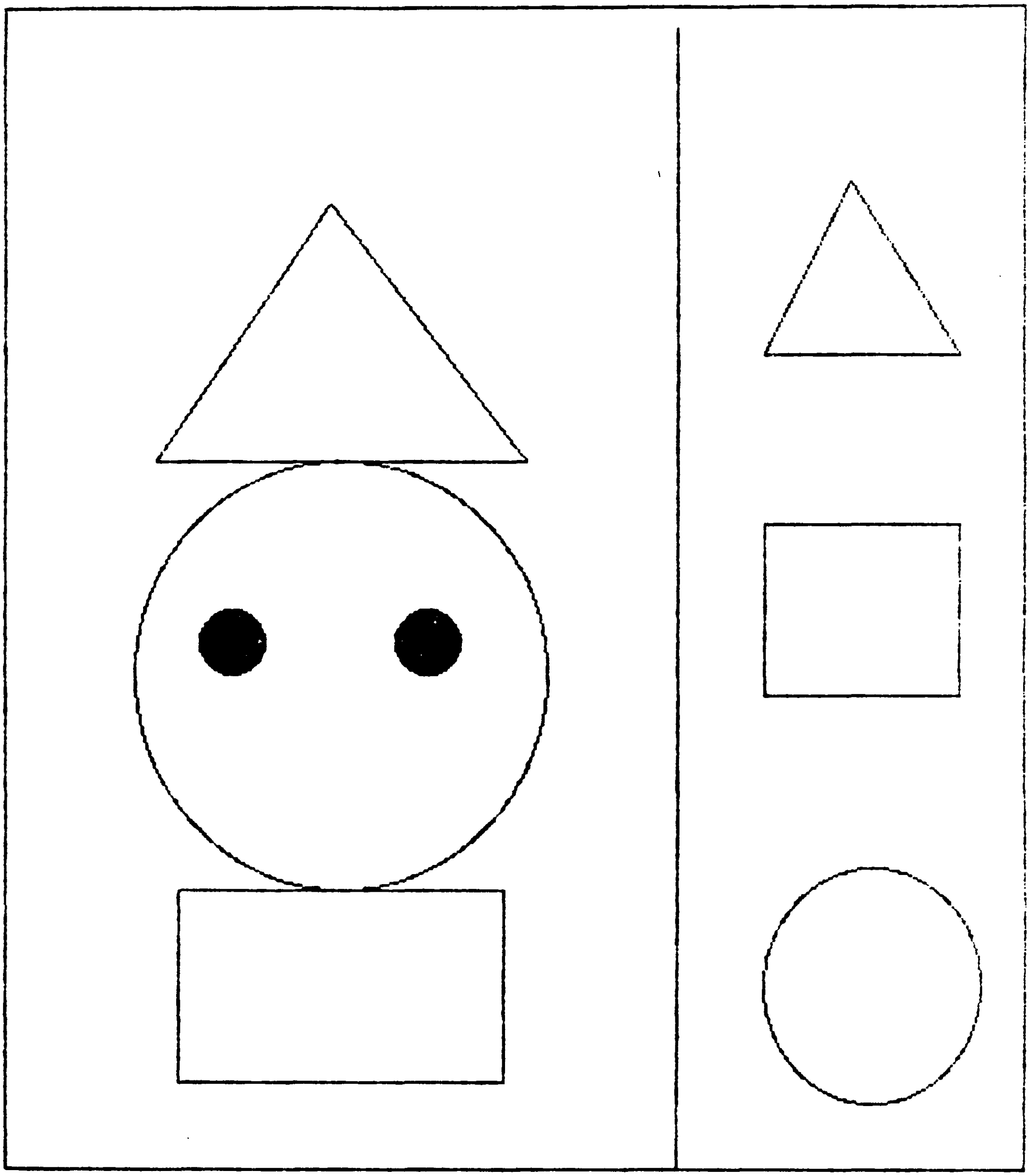


Figure 1.8: A Simple Graphical Editor

drag it into the work area; the **move** command which allows the user to move geometric shapes around the work area; and the **remove** command which allows any geometric shape to be removed from the work area. The corresponding task and control models describing this user interface using the notation described above is presented in Appendix A.1.

The next step in the design is the specification of the user interface from the high-level description provided by the control model. The specification language is based on state-transition diagrams and is influenced by SRI's SPECIAL language [23]. The specification comprises several state machines (called modules), where each machine is a collection of V and O functions. The V functions represent the machine state while the O functions describe transitions that change the machine's state. The specification of the example user interface is provided in Appendix A.1.

The following three tests are used to evaluate the user interface specification to determine its correctness.

[1] The specification is **consistent** if each operator in the control model is implemented by the functions in the specification.

[2] All the invariants in the task and control models must hold for the specification.

[3] The behavior of the specification is tested under designer determined conditions.

Probably the most impressive design methodology developed to date is Moran's Command Language Grammar (CLG) [24]. The basic premise of Moran's proposal is that

to design the user interface of a system is to design the user's model.

Thus a CLG representation of a system describes the user's conceptual model of the system.

The description of a user interface follows a top-down design methodology. The user's conceptual model is described, then a command language that implements the conceptual model is designed, and finally a display layout is designed to support the command language. The CLG representation is structured as a number of levels with the aim of separating the conceptual model of the system from its command language and to show the relationships between them. The six levels, each being a refinement of the previous level, are organized into three components which describe the CLG structure.

[1] **The Conceptual Component:** describes the organization of the system as abstract concepts. This component is comprised of the **task level** and the **semantic level.** The task level analyzes the user's requirements to specify the structure of the tasks which describes the system from the user's viewpoint. The semantic level defines the methods for accomplishing task structures in terms of the objects and operations (that manipulate these objects) around which the system is built. The semantic level serves both the user and the system; it describes the conceptual entities and operations for the user, and correspondingly the data structures and procedures for the system.

[2] **The Communication Component:** describes the command language and the dialogue of the user interface. The **syntactic level** and **interaction level** make up the communication component. The syntactic level is a further refinement of the semantic level, describing the command language with which the user communicates to the system. The methods of the semantic level are described in terms of the commands developed at the syntactic level. The meaning of the commands are defined in terms of the operations described at the semantic level. The command languages are described in terms of basic syntactic elements: commands, arguments, contexts, and state variables. The interaction level specifies the syntactic level elements in terms of physical actions, i.e., primitive device techniques for input (e.g., keypresses) and display actions for output. The rules describing dialogue structure are also described at the interaction level.

[3] **The Physical Component:** describes the physical devices of the user interface. The **spatial layout level** describes the nature of the system's display at each point of interaction. It therefore is concerned with the arrangement of input/output devices and the graphics facilities for displays. The **device level** describes the physical properties of input/output

devices and the underlying graphics primitives. These levels make up the physical component.

The command language grammar is the specification mechanism used in describing the first four levels that make up the conceptual and communication components. The levels comprising the physical component have not been developed yet. The CLG notation, informally described in Figure 1.9, is based on the concepts of frames [25, 26], schemata [27], semantic nets [28], and production rules [29].

To depict the complexity of defining the user interface using the CLG framework, the description of a user interface for a mail system is presented in Appendix A.2.

It is important to reiterate at this point that the above merely represents proposals of formal design methodologies for interactive system development. The formal interactive system model and its associated methodologies have the following serious drawbacks.

[1] The models and more importantly the design methodologies have not been completely developed nor have their effectiveness been tested.

[2] The formal model is as general as the expressive power of the underlying formalism used for the specification mechanisms. Most of the formalisms that have been used to specify user interfaces have not been very effective. Their use has either been applied to a restricted (textually based) set of user interfaces, or to only specifying the dialogue component of the user interface. Researchers have yet to develop a suitable formalism that can not only describe the complex behavior of user interfaces, but can also capture the inherently 2-dimensional nature of user interfaces. This lack of a suitable formalism may prove to be a serious drawback in describing the diverse nature of interactive systems, in general.

[3] Formal descriptions of systems are inherently complex. It is usually difficult for human factors experts, who are not computer scientists, to understand and effectively use these for-

CLG is a symbolic notation for describing systems as conceptual structures The basic objects of CLG notation are symbols and symbolic expressions. An expression, which is a structure of symbols, represents a concept by describing it, that is, by having its constituent symbols represent constituent aspects of the concept. An expression can be associated with a symbol as its definition The symbol can then stand for the expression and hence for its concept:

symbol → *expression* → *concept*

Notationally, symbols are arbitrary strings of characters, indicated by being in a sans serif typefont, like THIS. An expression is a list of symbols or subexpressions in parentheses For example, Q = (X Y (W Z)) defines the symbol Q by an expression with three elements—two symbols, X and Y, and a subexpression, (W Z), with two symbols. The syntax of expressions is not arbitrary. There are three forms of expression in CLG.

The first form is the basic CLG expression, a hierarchic description. It describes a concept by declaring that it is an instance of another concept, plus some modifications. For example, CHAR = (AN ENTITY CODE = (A NUMBER)) defines CHAR to be an instance of ENTITY with a component called its CODE, which is a NUMBER A descriptive expression thus consists of a prefix symbol, a type symbol, and a list of components A component of an expression may be thought of as a symbol definition within the localized context of the expression. The symbol naming a component may also be used in other expressions. A component CODE of an expression CHAR can be unambiguously referred to by the expression (THE CODE OF CHAR). The prefix symbols A or AN indicate that an expression is a pattern describing a class concept. The prefix THE indicates that the expression describes a unique referent.

There are some minor variations to the syntax of this form of expression: The prefixes A or AN may be dropped The symbol "=" is everywhere optional. A component may be unnamed. There is a special component, named OBJECT, that implicitly follows the type symbol. For example, (MOVE X FROM Y TO Z) has three components, the first of which, X, is the OBJECT of the MOVE.

The second form of expression is for representing collections of elements. It is indicated by a colon after the first symbol, which indicate the type of collection. For example (SET: X Y Z) and (SEQ: X Y Z) represent, respectively, a set and a sequence of three elements. XYZ = (ONE-OF: X Y Z) defines XYZ to be the disjunction of the three items.

The third form of expression, (* ...), contains any English statement It is usually used as a subexpression, allowing informal descriptions to be inserted anywhere within an expression structure

An expression represents a concept, and its components represent the parts or aspects of that concept Components can also represent the relations between expressions For example, if T = (A TASK) P = (A PROCEDURE), and M = (A METHOD FOR T DO P), then M shows the relation between T and P There are implicit relations between an expression and its component expressions. For example, the definition of P above should be (A PROCEDURE OF M), but the OF component does not need to be stated, since it is implicit The other implicit relation is IN. For example, if S = (SET: X Y Z), then X, Y, and Z implicitly have the component IN S.

Figure 1.9: The Command Language Grammar Notation

malisms. The example user interface specification presented in Appendix A.2 attests to the seriousness of this drawback.

[4] A working system is usually available only after a very complete specification of the interactive system. Therefore, it is difficult to involve the end users and human factors experts throughout the development process.

However, interactive systems are extremely complex and still not very well understood. There is need for more research to develop formalisms and methodologies suitable for interactive software engineering.

1.1.2. Practical Interactive System Models

Many researchers feel that the inherent properties of interactive systems - their complexity, their need to conform to many differing interfaces, and their frequent changes during their lifetime - necessitate a radical departure from the rigid formal approach. Thus, the **practical approach** adopted by many researchers is based on software engineering principles and techniques that have been proven, in practice, to be efficient and effective for the development of interactive systems. This has led to informal and more flexible models and methodologies for interactive software engineering.

What modern software engineering principles should be employed in building graphical interactive software systems?

As Brooks [30] points out, after almost two decades of software engineering research, the following principles have emerged as the most promising for interactive software development.

[1] **Rapid prototyping** of software, that is, the development of methodologies and tools that simulate the major interfaces and functionality of the intended software system. Brooks describes this principle as

> **the most promising of the current technological efforts, and one that attacks the essence, not the accidents, of the software problem.**

[2] **Incremental development,** that is, software should be grown, not built. Note that incremental development is tightly coupled with rapid prototyping.

The underlying principle of this **prototyping methodology** is to rapidly prototype a skeleton of the interactive system with an emphasis on the interaction aspects, and then **iteratively develop** or fine tune the interactive system. Such an approach has the following advantages over the more formal and rigid methodologies.

[a] The end users can be involved throughout the development process as a prototype of the system is immediately available.

[b] The prototype concept provides an appropriate framework to test out various approaches for the interaction aspect of the system quickly and efficiently.

[c] Iterative development ensures a **working system** at an early stage of the engineering process, thereby making it possible for the designers to always deliver the interactive system.

Modern software design techniques have provided the basis for the prototyping methodology. These design techniques can be summed up in two words: **encapsulation** and **reusability.** Encapsulation is a technique that minimizes the interdependencies between the modules that comprise the software system. Each encapsulated module defines an external interface which serves as a contract between the module and its users. This external interface defines the set of operations that provide the modules' functionality. All users can access the services of modules only through their external interface. To maximize encapsulation, implementation details of the module are suppressed in its interface. This allows modules to be reimplemented without affecting any of its users. Reusability is a technique that allows components that have been built and tested to be reused in the development of software systems. This technique is the main factor in the tremendous progress that has been achieved in computer hardware engineering.

One possible approach that combines the powers of encapsulation and reusability is the notion of **abstraction.** The behavior of an abstract object is described completely by a set of abstract operations defined on the object. Users of such abstract objects need not be concerned about how these abstract operations are implemented or on how the object is represented. Once these abstract objects have been defined, they can be implemented and integrated as part of the language (or more precisely the **specification mechanism**) that is used to design and develop software systems. Thus abstraction promotes both encapsulation as well as reusability. The base data types provided by high-level programming languages are the best examples of abstract objects. Programmers (re)use integers, reals, characters, strings, etc. solely on the basis of the operations define on them, with no concern for their implementation or representation. The provision of data types as the only abstract objects is precisely the factor that limits conventional high-level programming languages as an effective specification mechanism for the design and development of software systems.

The current emphasis of research in interactive software engineering is to identify, implement and integrate a large set of abstract objects in the specification mechanism, thereby providing software engineers with the necessary tools to effectively design interactive systems.

A popular model for interactive software systems that supports the prototyping methodology is presented in Figure 1.10 (adapted from [31]). In this model, an interactive system is built as three separate layers:

[1] the application layer implements the system's functionality,

[2] the user interface layer implements the user interface to the application layer allowing for interchangeable interfaces, and

[3] the virtual terminal layer implements the device-independent graphical primitives to facilitate the presentation of information to the user on any physical output device.

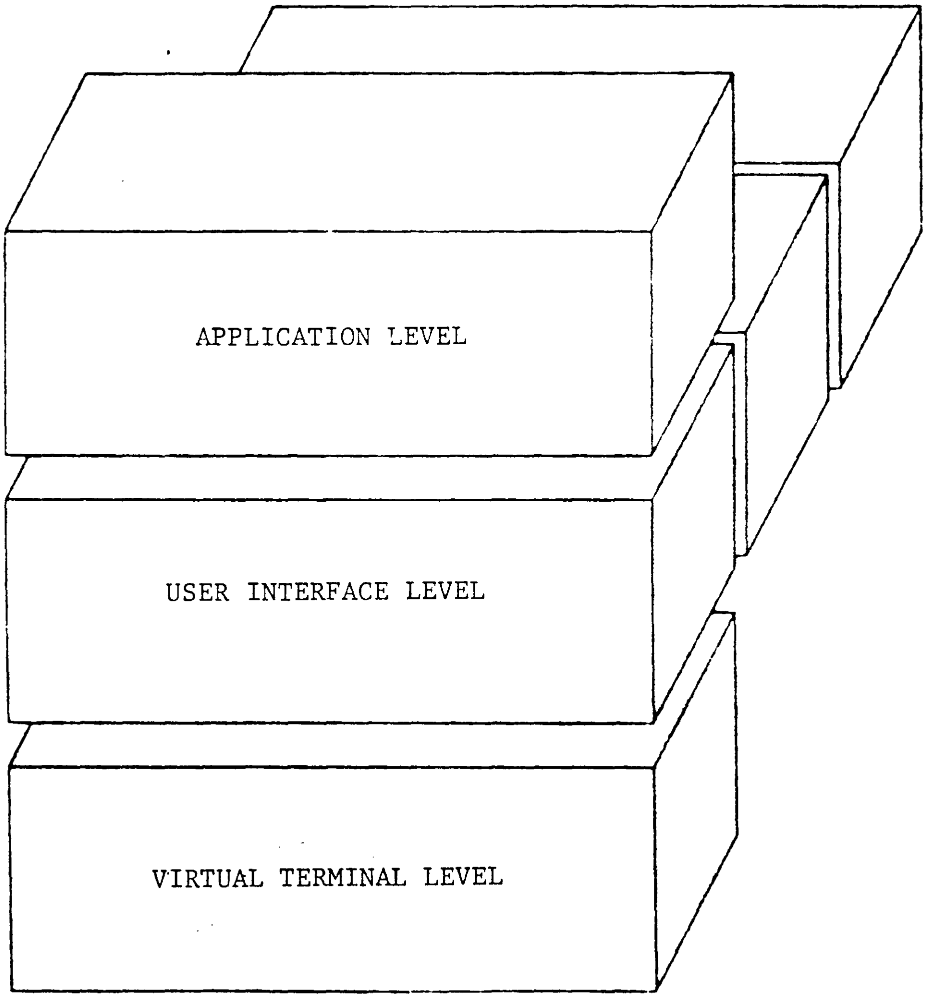


Figure 1.10: The Components of an Interactive System

The relationships between the components of the interactive system model is presented in Figure 1.11 (adapted from [31]). The important issues of this relationship are the following.

[a] The application holds all the information to be interfaced to the user.

[b] The interaction is controlled by the user through the user interface layer, that is the application is passive and the slave of the user interface. This allows the user interface to be detached and replaced as needed.

Research in **computer graphics** has resulted in powerful techniques, tools and standards that facilitate the efficient development of the virtual terminal layer. The design and development of the application layer has been the major concern of **software engineering** for over two decades, and is not only better understood but more importantly supported by a host of design methodologies and tools. The user interface layer is the least understood component of interactive software systems. It is, however, the most crucial component, as the user interface of interactive systems is the main factor in determining the effectiveness of the system.

It is important to point out that the interactive system model presented in Figure 1.10 supported by the prototyping methodology satisfies all the requirements of the ideal model presented above. Note that the concept of **interchangeable interfaces** facilitates total end user involvement in determining the behavior of the interactive system. Therefore, this ideal user-oriented interactive system model and its associated prototyping methodology will be used as a blueprint for the rest of this dissertation.

1.2. Developing Graphical User Interfaces

Current research in user interfaces is directed at two major issues:

[a] what properties constitute the **best** user interface, and

[b] how can user interfaces be effectively designed and developed efficiently.

The former issue falls in the realm of user psychology and human factors research. The latter issue is the current focus of software engineering, and is the main concern of this dissertation.

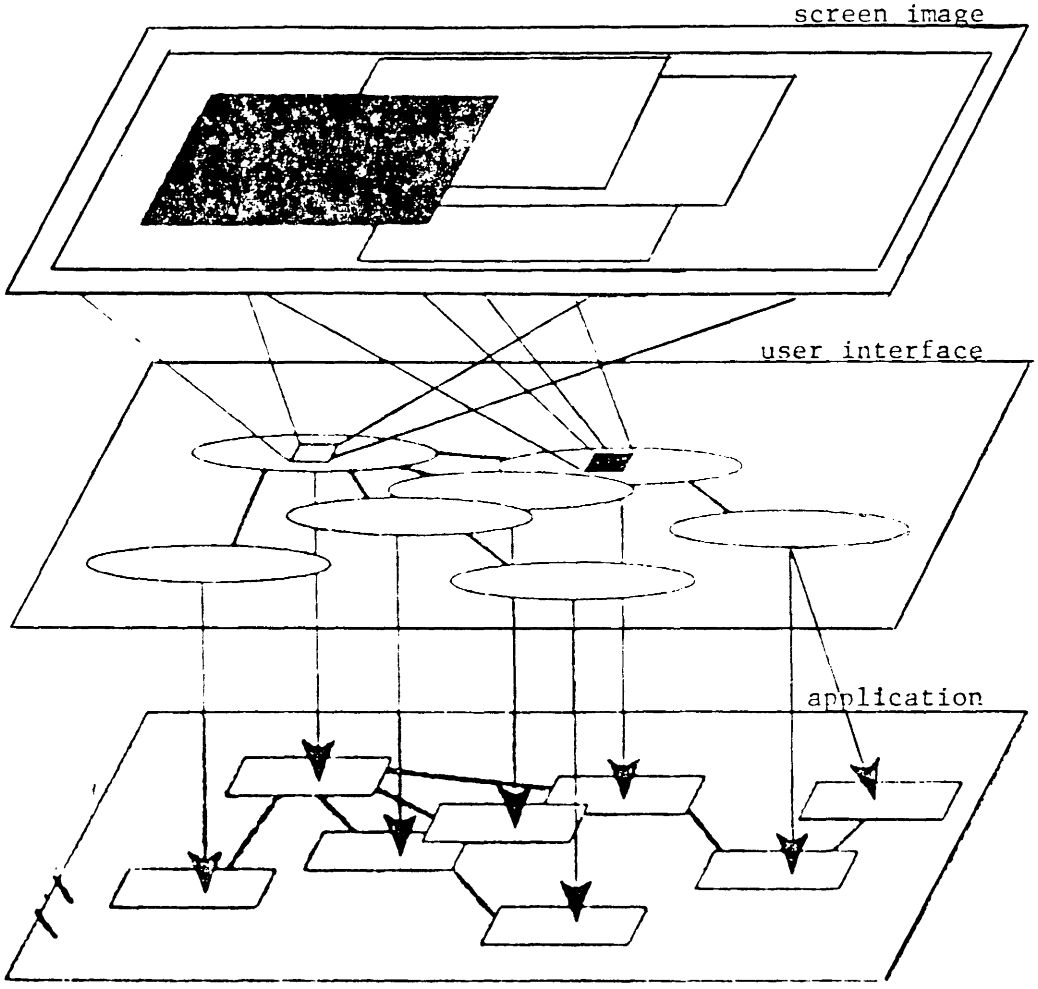


Figure 1.11: The Relationships between Interactive System Components

Following the model of interactive systems as presented in Figure 1.10, the functionality of the user interface component is defined by the following:

[1] accepting user input, that is, function invocation and parameter specification,

[2] communicating user requests to the application, and

[3] displaying the application's internal information base to the user.

Fundamental to the interactive system model is the complete separation of the user interface component from the application component. This separation not only facilitates the development of effective user interfaces, but more importantly provides the necessary basis for interchangeable user interfaces.

Why is it necessary to facilitate interchangeable user interfaces? There exists two opposing models in any interactive software system.

[a] **The System's User Interface Model:** This is the model that defines the underlying principles on which the system's user interface is built from the perspective of the system (application) designer.

[b] **The User's System Model:** This models the user's perception of how the system works.

The effectiveness of an interactive system can be measured in terms of the **distance** between these models. For novice users, most interactive systems have a distance much greater than zero; that is, the tool does not behave as the user expects. Obviously, an effective interactive system achieves a distance that is much less than or equal to zero; that is, the system behaves exactly as the user expects, or better.

There are two approaches to minimize this distance.

[1] Force the user's model to move towards the system's model. That is, force the user to learn the particular user interface developed for the interactive system. This **system-oriented** approach is adopted by most of the currently available interactive systems.

[2] Allow the system's model to move towards the user's model. This **user-oriented** approach allows the user to define the behavior of the system.

The above two approaches identify the extreme ends of the spectrum of possible approaches for developing user interfaces. The system-oriented approach provides facilities to the designer to develop the best interface for a given application. The user-oriented approach provides facilities to the end user to develop interfaces for a given application that are customized according to the end user's working habits and experience. Note that interfaces developed with the system-oriented approach can be pushed towards the user-oriented approach through mechanisms that allow the interface to be altered. However, such flexibility is usually restricted to particular aspects of the user interface, and more importantly the choices for altering the interfaces must usually be hard-wired into the interface. Most of the current interactive systems provide flexibility in their interfaces at the level of color, key settings and user defined representations of internal (application) objects.

Since it is very difficult to generalize the peculiarities of a diverse user population to arrive at the **best** possible user interface, the user-oriented approach is better suited in building effective user interfaces. It is also important to note that the user's model of the system changes tremendously with experience, that is, as the user becomes an expert with the system. Thus, to maximize user productivity, it is very desirable to allow the user interface to evolve with the user's experience. The above has identified the ultimate goal of user interface research: allowing any user to develop a personalized interface for any interactive software system, thereby maximizing both the effectiveness of the interactive system as well as the productivity of the user.

A large percentage of user interface research is therefore aimed at building **User Interface Development Environments (UIDE).** UIDEs have the following characteristics.

[1] They automate the development of a large portion of the interactive software.

[2] They (usually) allow a declarative specification of the interactive system.

[3] They provide the necessary basis for rapid prototyping and incremental development of the system.

[4] They provide a test-bed for comparing possible approaches to the development of the system.

Most of the UIDEs available today have been built to support the system-oriented approach, that is, the provision of facilities to allow the user interface (interactive system) designer to develop the best user interface for a given application. Minimal research has been directed at building UIDEs in support of the user-oriented approach. The major aim of the research presented in this dissertation is to develop a general-purpose user-oriented UIDE.

There are several important differences between system-oriented and user-oriented UIDEs.

[1] **Single vs. Multiple Interfaces:** The system-oriented UIDE provides facilities for the development of a single user interface for a given application. In contrast, the user-oriented UIDE must provide for the development of multiple interfaces for a given application. This results from the requirement of allowing end users to define the behavior of the interactive system.

[2] **Designer vs. End User Interface Definition:** Interface definition in a system-oriented UIDE is performed exclusively by the interface designer. End users may be allowed to alter interfaces if the designer hardwires the possible alternatives in the user interface. The main goal of the user-oriented UIDE, on the other hand, is to provide to the end user, facilities similar to those provided to the designer by system-oriented UIDEs.

[3] **Static vs. Dynamic Binding:** The user interface that is developed within a system-oriented UIDE is bound to the application to form the interactive system. This early binding of the interface to the application does not allow for interchangeable interfaces within the ideal interactive system model presented in Figure 1.10. However, a user-oriented UIDE

must provide for dynamic binding of the user interface to the application to facilitate definition and/or modifications by the end user.

[4] **Coupling vs. Complete Separation of Interface and Application:** A system-oriented UIDE need not be concerned with the complete separation of the user interface from the application, since the major focus is to package these components into an interactive system. Such a complete separation, however, forms the basis for user-oriented UIDEs to facilitate interchangeable interfaces.

The many user interface generators that have been developed to date employ radically different philosophies to user interface development. There exists, however, no general classification frameworks for UIDEs. Classification frameworks are an important vehicle in determining the common characteristics of UIDEs as well as differentiating between UIDEs. The classification scheme that is proposed below draws an analogy from Flynn's classification of computer architectures [32]. Since the essential computer process is the execution of a sequence of instructions on a set of data, computer architectures can be classified into the following four frameworks according to the multiplicity of instruction and data streams that can be handled:

+ Single Instruction stream - Single Data stream (SISD)
+ Single Instruction stream - Multiple Data stream (SIMD)
+ Multiple Instruction stream - Single Data stream (MISD)
+ Multiple Instruction stream - Multiple Data stream (MIMD)

In the case of UIDEs, the essential process is the development of a user interface for an application domain. Thus, UIDEs can also be classified into the following four frameworks according to the multiplicity of user interfaces and application domains that can be handled:

+ Single user Interface - Single application Domain (SISD)
+ Single user Interface - Multiple application Domains (SIMD)
+ Multiple user Interfaces - Single application Domain (MISD)
+ Multiple user Interfaces - Multiple application Domains (MIMD)

It is important to note that the first two frameworks support the system-oriented approach while the latter two frameworks support the user-oriented approach to developing user interfaces. The following four subsections provide a discussion of each of the four frameworks for UIDEs.

1.2.1. The SISD Framework

The major emphasis of UIDEs within the SISD framework is the development of a single interactive system with its user interface and application components. All users interact with the system through the single user interface as depicted in Figure 1.12. This approach is currently popular as it allows **good** user interfaces to be built for existing applications to form interactive systems which can be immediately incorporated into the host operating environment. If, however, the entire interactive system is built using the traditional approach of software engineering, then this approach is also the most costly. As pointed out in the introduction, such an approach is precisely what modern software engineering research is trying to avoid.

Since the major concern of this class is the development of a particular user interface for a particular application, the role of a UIDE must be expanded to include facilities for the development of the interactive system as a whole. The major concern of SISD UIDEs is to extend classical software engineering to provide mechanisms for the user interface component that are consistent with, and as powerful as, those provided for the application component.

Since a large percentage of traditional software engineering is based on formal specification mechanisms for each phase of the life cycle, SISD UIDEs benefit most from the formal approach to interactive software engineering. The main focus of research in SISD UIDEs is therefore the development of suitable formalisms for the specification of the user interface component. Furthermore, to be cost effective, SISD UIDEs must also provide mechanisms that automate the generation of the user interface from the formal specification, as well as managing the user interface throughout the development process.

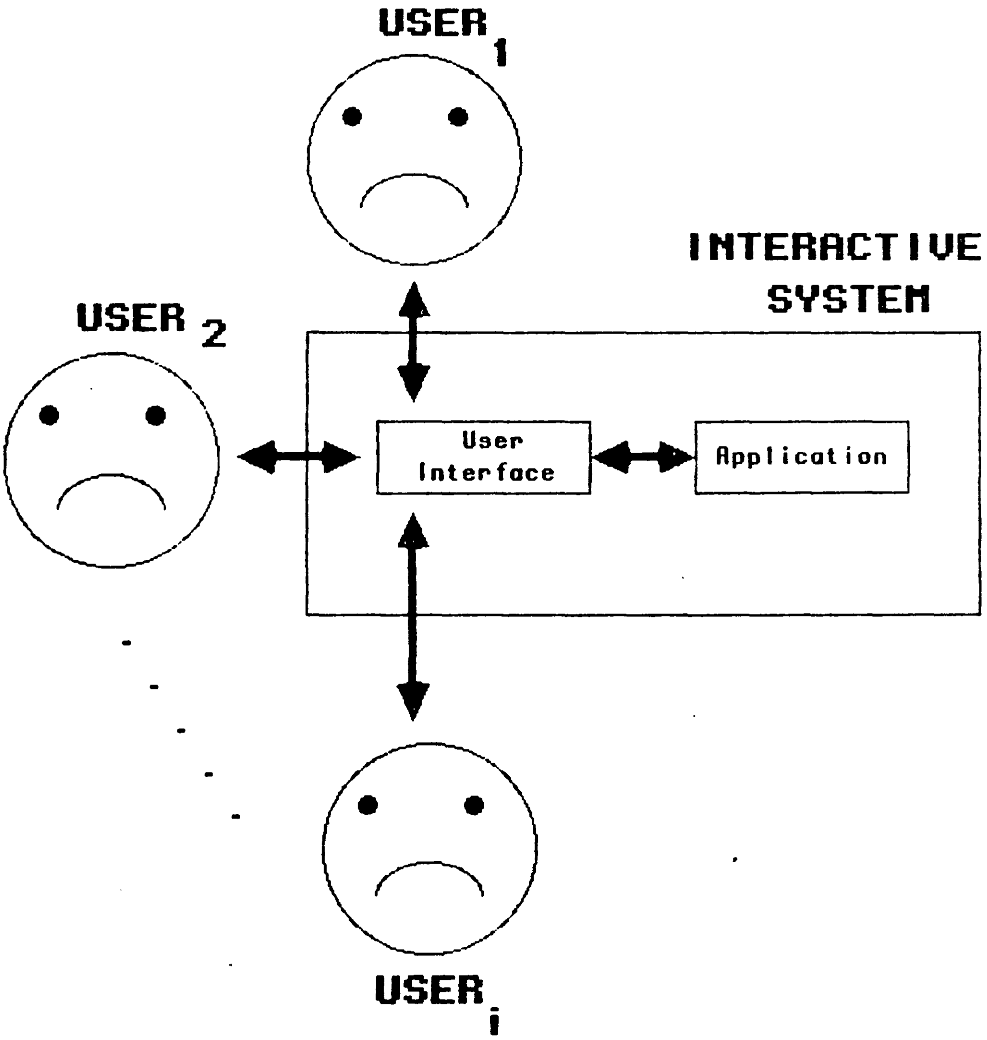


Figure 1.12: Interacting with a Single User Interface

1.2.2. The SIMD Framework

The major effort in the class of SIMD UIDEs is to provide a consistent user interface development framework for a variety of application domains. The effectiveness of the SIMD approach is due to the fact that users need only familiarize themselves with the user interface framework once. Thereafter, users can easily use all other interactive systems that present a similar user interface framework. This approach is currently the most cost effective as it makes an entire interactive operating environment immediately accessible to a large user population. The users' perspective of the SIMD environment is depicted pictorially in Figure 1.13.

The SIMD approach has been popularized initially by the introduction of personal computers and more recently by high powered graphical workstations. These personal environments necessitate consistent user interface frameworks to be cost effective. The standardized user interfaces provided by such environments are characterized by the following.

[a] A bitmapped display screen contains one or more overlapping rectangular areas called **views (windows)** which presents information to the user.

[b] Function invocation and parameter specification is performed exclusively by selection either through keys or a mouse, and extensive use of menus.

[c] The keyboard is used to input textual information to the interactive system.

The specification of the user interface usually consists of the description of the information that is displayed to the user and the function and parameter names that form user requests to the interactive system. The way in which the information is displayed and the manner in which user requests are input is usually predetermined and standardized. Therefore, the major emphasis of SIMD UIDEs is to provide the necessary environment and tools to rapidly prototype and iteratively develop the user interface of an application. The practical approach to interactive software engineering is most appropriate to SIMD UIDEs.

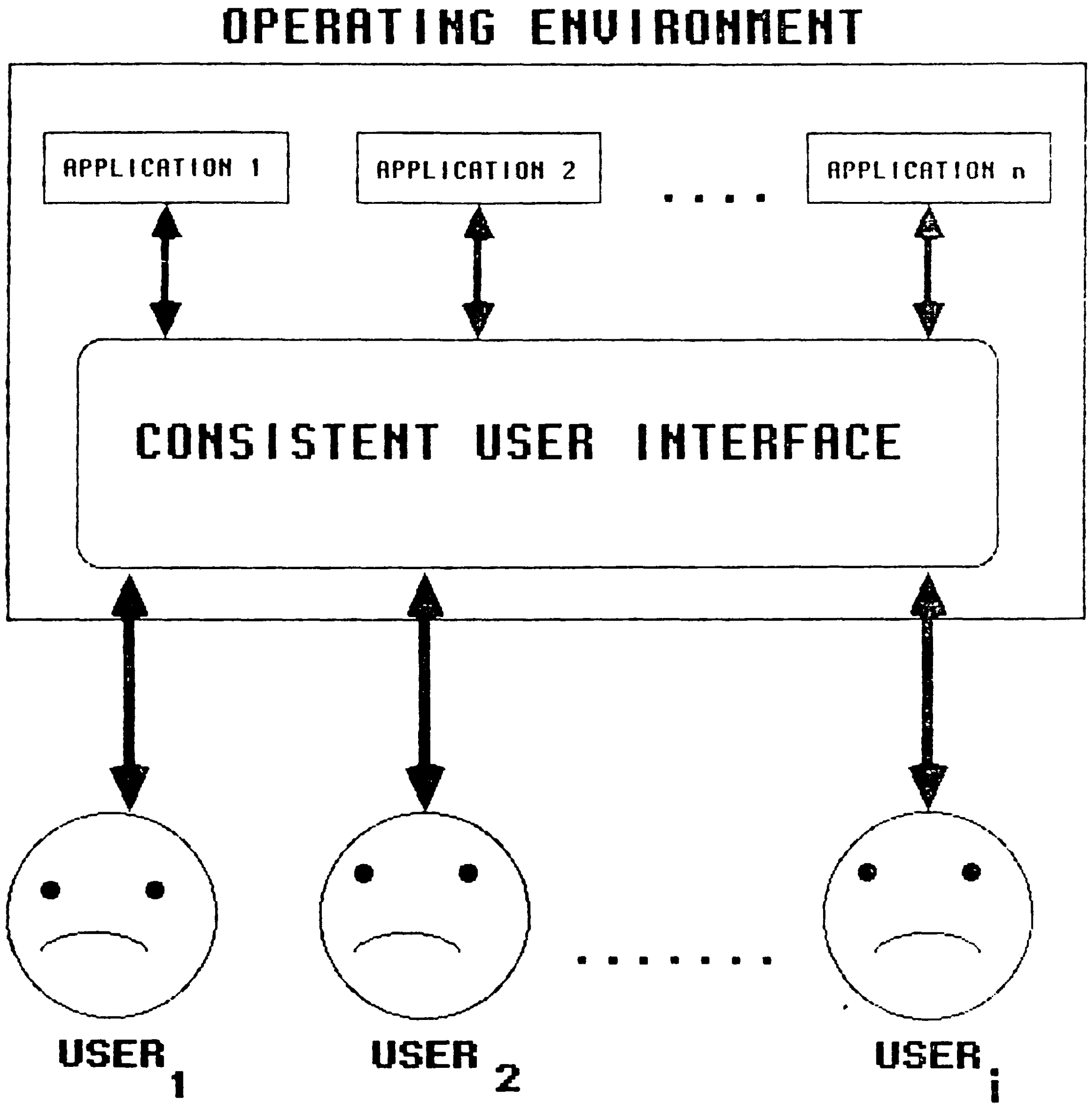


Figure 1.12: Interacting in a SIMD System

Even though standardized consistent user interfaces give cost effective access to an entire interactive environment, they can also seriously hamper user productivity and the effectiveness of the interactive systems as users become experts. Another major problem with the SIMD approach is that any major technological advances in user interface techniques would necessitate a redefinition of the entire interactive environment. In other words, what is considered as the best user interface today could very well be technologically obsolescent in a few years. Thus, some of the major concerns of SIMD UIDE research is to provide flexible mechanisms that allow

[a] the user interfaces to evolve with the user's experience, and

[b] newer techniques to be incorporated into the environment.

1.2.3. The MISD Framework

The main concerns of MISD UIDEs is to allow the generation of personalized user interfaces for users interacting with a single application. The main advantage of MISD UIDEs is that they increase the effectiveness of the tool by (potentially) allowing each user to develop a user interface to suit his/her personal working habits thereby maximizing user productivity. The environment presented to the end users by the MISD approach is depicted pictorially in Figure 1.14.

The main difference between the SISD and MISD frameworks is that user interface development facilities are provided to the interface designer by the former and to the end users by the latter. Since both frameworks are concerned with a single interactive application, the MISD framework is also suited for the formal approach to interactive software engineering. That is, user interfaces are generated through formal specification mechanisms. However, the central theme of the user-oriented approach dictates that MISD UIDEs allow computer science naive users to specify user interfaces. This is the major drawback to the effectiveness of using formal specification mechanisms at the end user level.

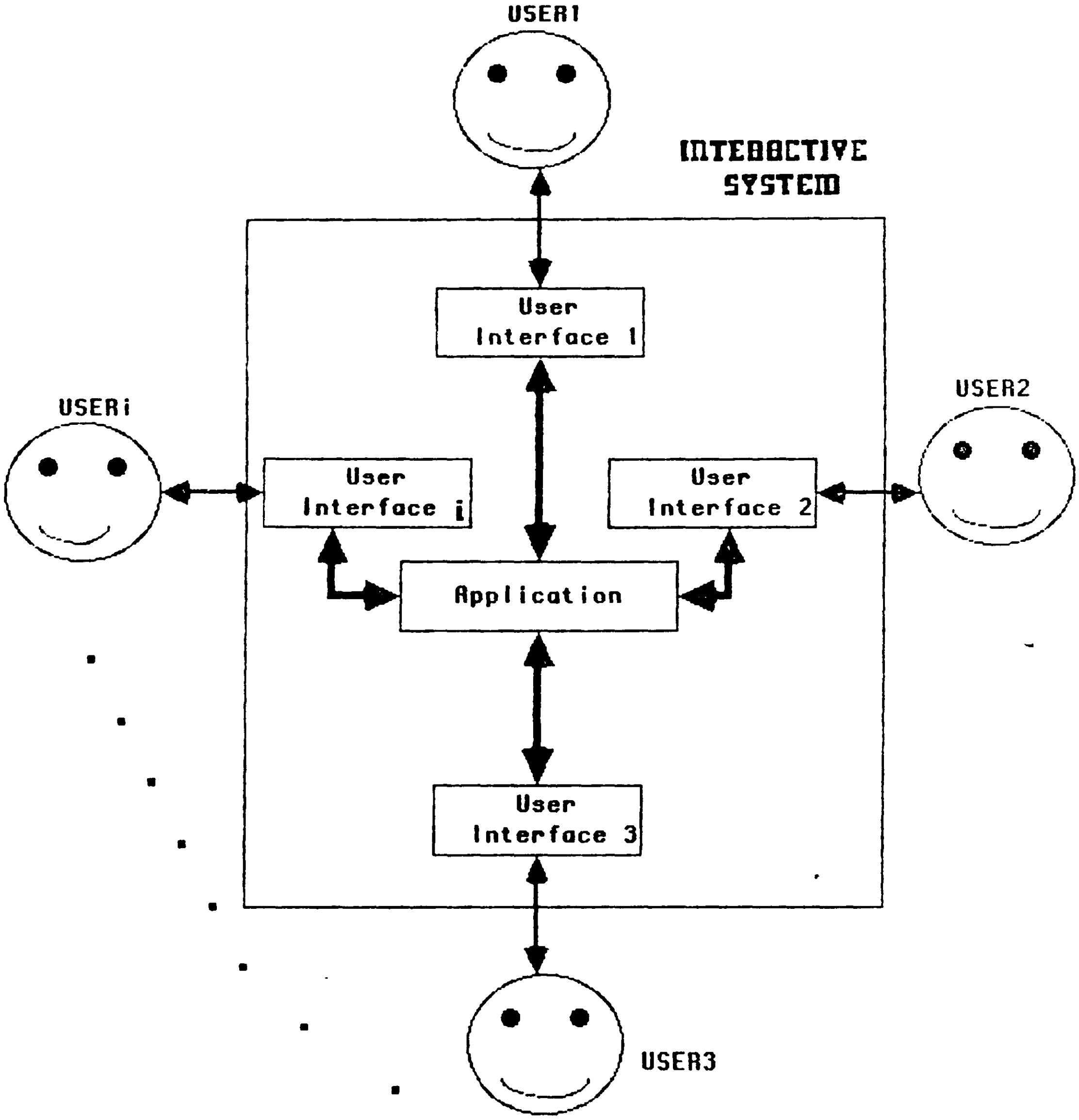


Figure 1.14: Interacting in a MISD System

The major problem with the MISD approach is cost, as it necessitates the development of an entire user interface generator system for every interactive application. The MISD approach could be cost effective only if user interface research progresses to a point that facilitates the development of a system that could generate MISD UIDEs. Such a system would accept a specification of the interactive application and generate a MISD UIDE for the particular application! However, such an approach is a special case of the MIMD approach presented below.

1.2.4. The MIMD Framework

The basic premise of the class of MIMD UIDEs is that both user productivity and the effectiveness of interactive operating environments can be maximized if users are able to develop personalized interfaces for any interactive system. The MIMD approach not only facilitates the development of interfaces to suit the peculiarities of each user, but more importantly allows the user interface to evolve with the user's experience. The user environment resulting from the MIMD approach is depicted pictorially in Figure 1.15.

It is obvious that the development of MIMD UIDEs is the ultimate goal of user interface research. However, the reality is that no MIMD UIDEs have been developed to date, attesting to the relative infancy of user interface research. The major problem in developing MIMD UIDEs is that it necessitates a complete separation of the user interface component from the application component. Note that UIDEs under the previous three frameworks tightly couple the two components.

What are the main problems in facilitating this complete separation of the user interface from the application?

[1] Developing a model that generalizes interactive applications to allow the UIDE to handle multiple application domains.

[2] Developing a model that generalizes function invocation and parameter specification within the application so that the user interface can communicate user requests to the application

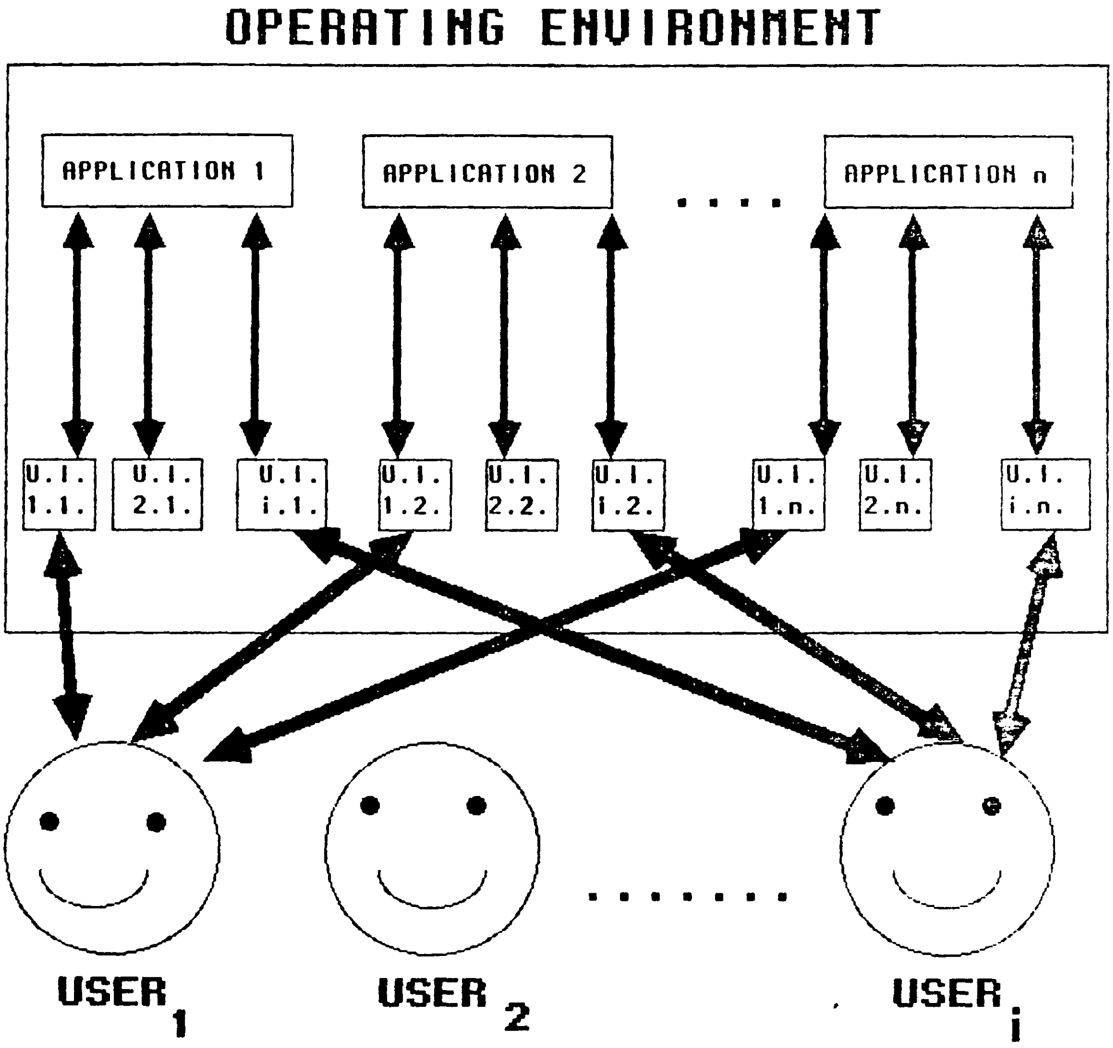


Figure 1.15: The MIMD User Environment

in a consistent manner.

[3] Developing a model that generalizes the flow of information (that is to be presented to the user) from the application to the user interface.

Thus, the MIMD approach necessitates a radically different approach to the design of the entire interactive system, both the application component as well as the user interface component. This is the main reason why the MIMD approach has not occurred in practice. This dissertation develops a design framework that demonstrates the feasibility of MIMD UIDEs.

1.3 The Aim of This Research

A major goal of this research is to allow any user to develop a personalized interface for any interactive application, that is, the development of an **MIMD user interface development environment.** Fundamental to the development of such a UIDE is the complete separation of the user interface component from the application component. This separation necessitates devising less tightly coupled models of the application and user interface than has been reported to date. The main features of the MIMD model developed in this dissertation to achieve the desired degree of separation are as follows.

[1] Interactive applications are modeled as editors providing a set of functions that manipulate 2-dimensional graphical objects.

[2] **Interactive data structures** are introduced for maintaining and manipulating both the internal and external representation(s) of application information as a single unit. These external representations form the basis for presenting internal information to the user.

[3] Since interaction with the user must be the sole responsibility of the user interface component, function interaction is modeled as follows. Application functions are modeled as a set of **services.** Each service processes a (set of) parameter(s) independently. For each service in the application, a corresponding **service interface object** is defined in the user interface component. The service interface object interacts with the user to specify the

required (set of) parameter(s), calls the associated service within the application, and displays the result of the service to the user.

Using the above model, the user interface component is modeled to allow personalized specifications at all levels; including the internal entities of the interactive system, the characteristics of the display of information, and the interaction tasks, techniques and devices used for parameter specification.

1.4 Organization of the Dissertation

The rest of this dissertation is organized as follows. Chapter 2 presents a survey of the literature on user interface software systems. This survey concentrates on user interface development environments under the SISD, SIMD and MISD frameworks. Chapter 3 develops an appropriate model for a user interface development environment under the MIMD framework. The main concern of this model is the complete separation of the application from the user interface. Appropriate models for both the application and user interface components to support such a complete separation are developed. The design of the MIMD UIDE is developed in Chapter 4. The design is carried out using the **object-oriented** paradigm, as this is currently the best available paradigm that supports the methodology of rapid prototyping. The main focus is the design of objects involved in presenting information to the user as well as function invocation and parameter specification. A prototype implementation of the MIMD UIDE based on the design developed in the previous chapter is presented in Chapter 5. This implementation is carried out in the Smalltalk object-oriented environment. Finally, Chapter 6 concludes this dissertation and suggests possible further research. After presenting the major contributions of this research, it is shown how the framework that has been developed can be used as the basis for generating interactive tools which will ultimately lead to the automatic generation of interactive environments.

CHAPTER 2

USER INTERFACE DEVELOPMENT ENVIRONMENTS

The computing breakthroughs of the past such as high-level languages, time-sharing, and programming environments (e.g., Unix and Interlisp), and the current research efforts in Ada, object-oriented programming, artificial intelligence, expert systems, automatic programming and visual programming, attest to the continuing desire of researchers to find the **silver bullet** that will defeat the software engineering monster [30]. In their quest for the silver bullet, researchers have directed their efforts in building **Software Development Environments (SDEs),** which provide an **integrated** set of tools/facilities dedicated to the development of software systems.

This chapter presents a survey of the research in building environments to develop user interfaces, that is, **User Interface Development Environments (UIDEs).** In the previous chapter, UIDEs were classified into four distinct categories based on two broad criteria. The first criterion distinguishes between the underlying approach for software development. The two extremes of possible approaches were identified as **formal** and **practical.** To reiterate, the formal approach relies on a (formal) specification language as the basis for software development. The intended software is described using this specification language. This description is validated to ensure correctness and consistency, and then translated into a working prototype of the system. The environment provides the facilities to test, validate and fine tune the prototype into a working system. The practical approach relies on a set of tools that provide facilities to develop, test and validate separate modules of the intended software. These individual modules are then integrated into a comprehensive system. Additional facilities allow the system to be tested and modified as a whole.

Due to the inherent interactive nature of user interfaces, the second major criterion distinguishes among the intended users of the development environment. If the environment is designed for the interactive system developer, then the UIDE is **system-oriented.** On the other hand, a **user-oriented** UIDE is concerned with providing development facilities to both the system developer and the end user of the interactive system.

The UIDEs that have been developed to date fall into the first three categories; that is, the SISD, SIMD and MISD frameworks. The next three subsections present a detailed survey of the UIDEs that have been developed within each framework.

2.1. SISD Framework: Language-Based Development Environments

UIDEs within the SISD framework support the development of a single interactive system with its user interface and application components. The discussion that follows will, however, concentrate exclusively on the support that is provided for the user interface component.

The major premise of UIDEs within the SISD framework is the provision of an appropriate formal specification mechanism to precisely describe the user interface. To be effective, the formal specification mechanism must have the following desirable characteristics:

[1] The specification mechanism should be mainly concerned with describing the user's model of the system. Thus, its constructs should represent concepts meaningful to the user, e.g., replying to a message, editing a file, etc.

[2] The specification must be precise, describing the exact behavior of the system for each user input.

[3] The specification must be concise and easy to understand; that is, it must be a productive tool for the designer.

[4] The specification must allow for verification of the user interface and also allow checking for consistency.

[5] The specification mechanism must be powerful enough to describe non-trivial user interfaces with minimum complexity.

[6] The specification should allow for automatic evaluation of user interfaces according to specified human factors.

[7] The specification mechanism should allow for experimentation and verification of human factors principles to test different design alternatives.

[8] The specification mechanism must separate the functionality of the system from its implementation. It should describe the behavior of the user interface without containing the method of implementation.

[9] The specification mechanism must allow a prototype system to be generated directly from the specification.

From the inception of user interface research in the late 60's, scientists have yet to develop a suitable formalism that can not only handle the complexity of describing user interfaces but also have the desirable properties listed above. The formalisms that have been developed to date have been used either to describe textual interfaces or to describe a particular component (usually the dialogue component) of the user interface. To get a good understanding of the complexity of the problem, it is instructive to trace the progress that has been made in the quest for a suitable formalism.

In the early 70's, the language model of interaction (presented in section 1.1.1) was popular. To reiterate, the language model views interaction as a conversation between man and machine. This conversational language consists of two components, the input and output language. Each component has an associated semantic, syntactic and lexical component. These are integrated together by the conceptual model of the system. The popularity of the language model can be attributed to the vast knowledge that had been accumulated on formalisms (such as context-free grammars and finite-state diagrams) that were developed in support of language and automata theory. The strategy was to extend these existing formalisms to make them suitable for describing the input and output languages of interaction.

In the next decade, it became obvious that these formalisms were not capable of handling the complexity of describing interaction. However, experience showed that they could be appropriately extended to describe the input language (that is, the dialogue) of interaction. By the mid 80's, it was realized that the language model needed modification to accommodate separate formalisms for each of the components of interaction. This effort resulted in the Seeheim model [33] of user interfaces, as presented in Figure 2.1.

The Seeheim user interface model comprises the following three components.

[1] The **presentation component** is the lexical level of the user interface and is therefore concerned with the physical representation. This includes input and output devices, screen layout, interaction techniques, and display techniques.

[2] The **dialogue control component** is the syntactic level of the user interface, and is responsible for the structure of the commands and dialogue used by the user.

[3] The **application interface model component** is the semantic level of the user interface, which defines the interface between the user interface and the application.

Within the context of the Seeheim model, the basic premise of UIDEs is to provide suitable formalisms to describe each component of the user interface. All of the formalisms that have been developed to date have been applied only to the dialogue control component.

The next subsection provides an in depth discussion of the formalisms that have been developed for the dialogue component. The following subsection describes the UIDEs that have been developed based on these formalisms. These UIDEs are commonly known as **User Interface Management Systems (UIMSs)** in the literature.

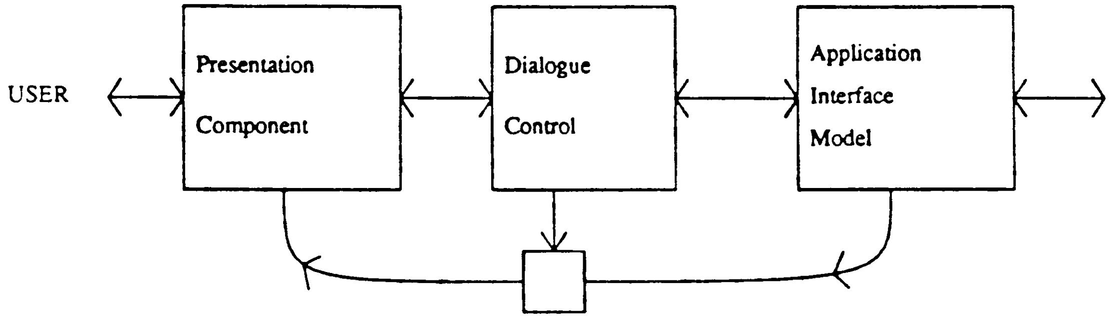


Figure 2.1: The Seeheim Model of User Interfaces

2.1.1. User Interface Specification (Dialogue) Models

The dialogue specification models that have formed the basis of most of the currently available UIMSs can be categorized into the following:

[1] **Grammar-based Models:** These models are based on formalisms that were developed for the specification of textual programming languages. Most grammar-based UIMSs use modified context-free grammars that are appropriately augmented to facilitate interaction specification.

[2] **Finite-state Models:** These models are based on state transition diagrams, and include simple, recursive and augmented transition networks.

[3] **Event Models:** These models are based on the concept of events that have been formulated as a generalized facility to handle input processing within graphics packages. Input devices are viewed within such packages as event generators. These events are placed in a queue, and can be processed by the application. For example, a tablet generates an event every time it is moved and when its buttons are pressed or released. The event dialogue model extends the above basic idea to an arbitrary number of event types and an event handler for each type. When an event is generated, it is sent for processing to the appropriate event handler.

The next three subsections present a detailed discussion of the above three dialogue models. The following subsection presents an analysis of the three models in an attempt to determine the **best** model, where best is measured in terms of descriptive power as well as usability.

To complete the discussion of dialogue models, it is important to point out another popular model that is based on the notion of data abstractions, which

> **comprises a group of related functions or operations that act upon a particular class of objects, with the constraint that the behavior of the objects can be**

observed only by the application of operations. [34]

Data abstractions are exemplified by the types provided by general purpose programming languages, e.g., integer, real, character, string, record, etc. Such data types are described precisely by a well defined set of operations for manipulating objects of the type. This concept has also been applied to describe other data structures, such as lists, trees, tables, etc. Many specification techniques have been used for data abstractions [35]. A popular technique is the algebraic specification technique [36] that specifies the relationships between operations using algebraic equations. Algebraic specifications have two major advantages, their numerical applicability and the existence of heuristics for writing consistent and complete axiomatizations. The concept here is to specify the types of data needed to support user interfaces and describe precisely the operations to manipulate the data. Examples of data that is relevant to user interfaces might be screens, windows, menus, message, etc. The designer is presented with a programming language or system that supports the user interface data types and their associated operations. The designer creates instances of these data types and employs the operations provided to describe a user interface.

However, axiomatic specification techniques have only been used for the paper specification of user interfaces to prove their completeness, and more importantly as the basis for precisely describing graphical programming languages [37]. Appendix B presents the relevant research on axiomatic models as concerns user interface specification.

2.1.1.1. Grammar-Based Models

The language model of interaction provides the motivation for using grammars to describe human-computer dialogues. The classical context-free grammar is augmented to allow for the specification of actions performed by the computer (that is, the application). Formally, a context-free grammar model, G, for dialogue specification, is a 5-tuple **G = (N,T,R,P,S)** where

[1] **N** is a finite set of symbols called nonterminals;

[2] **T** is a finite set of symbols called terminals, and there is one symbol in **T** for each of the input tokens produced by the presentation component;

[3] **R** is a finite set of symbols that correspond to the actions attached to the productions;

[4] **P** is a finite set of productions of the form

$$\mathbf{n} \dashrightarrow \mathbf{x},\ \mathbf{r},$$

where **n** is a member of **N, x** is a member of **(N U T)*,** and **r** is a member of **R;** and

[5] **S,** a member of **N,** is the start symbol for the grammar.

If **G** describes the dialogue control component of a user interface, then the language **L(G)** contains the legal sequences of user actions. A context-free grammar **G** can be used to produce a parser, which recognizes the user actions in **L(G).**

In practice, UIMSs have used context-free grammars to describe only the input language, that is, the language used by the user to communicate with the computer. The output language, that is, the language used by the computer to present information to the user, is difficult to specify with context-free grammars and is usually described by some other means.

The context-free grammar model was initially used to describe command languages for text-oriented interfaces. In [38], examples of using context-free grammars to specify two commands of a text-oriented interface are presented. Figure 2.2 illustrates the specification for a "login" command. This command requests the user's name, a password and a security level. The user is asked to reenter the name until recognized, is allowed two tries for a correct password, and cannot enter a security level that is higher than the security clearance assigned to the user (the default security level "unclassified" is assigned if the user's request is not appropriate). In Figure 2.2, lower case symbols denote nonterminal symbols; upper case are terminal symbols; production rules are annotated with conditionals; system responses and actions are specified within square brackets. A condition must be satisfied at the input stream corresponding to its position in the production rule, for a production to match. When a match occurs, the

```
Login::=        badpw* goodpw [resp: "Enter security level"] getseclevel

badpw::=        loguser onetry PASSWORD [cond: $PASSWORD≠GETPASSWD_USER($USER)
                        resp: "Incorrect password--start again"]

goodpw::=       loguser PASSWORD [cond: $PASSWORD=GETPASSWD_USER($USER)]
|               loguser onetry PASSWORD [cond: $PASSWORD=GETPASSWD_USER($USER)]

loguser::=      LOGIN [resp: "Enter name"] getuser [resp: "Enter password"]

getuser::=      baduser* USER [cond: EXISTS_USER($USER)]

baduser::=      USER [cond: not EXISTS_USER($USER) resp: "Incorrect user name--reenter it"]

onetry::=       PASSWORD [cond: $PASSWORD≠GETPASSWD_USER($USER) resp: "Incorrect password--reenter it"]

getseclevel::=  badsl* [resp: "Your security level is Unclassified"
                        act: CREATE_SESSION($USER,$PASSWORD,Unclassified)]
|               badsl* SECLEVEL [cond: $SECLEVEL<=GETCLEARANCE_USER($USER)
                        act: CREATE_SESSION($USER,$PASSWORD,$SECLEVEL)]

badsl::=        SECLEVEL [cond: $SECLEVEL>GETCLEARANCE_USER($USER)
                        resp: "Security level too high--reenter it"]
```

Figure 2.2: Grammar Specification of a LOGIN Command

response (if any) is displayed and the specified action (if any) performed. The special token NULL represents no input, symbols followed by a "*" denote one or more instances of that symbol, and a symbol preceded by "$" denotes the current value for the specified input token. Figure 2.3 presents a grammar specification of a "reply" command which allows the user to send a reply to a message that has been received. The reply command is made up of an optional message identifier (default is "CurrentMsg"), the text comprising the reply, and an optional list of additional addressees (which consists of the word "To" or "Cc" depending on how the reply should be addressed, followed by one or more addressees).

The above examples use grammars to specify only the input language of the user interface. They do not however specify how the output language is formed, that is, how computer responses are generated. Thus, a more precise description uses grammars not only to parse the input but also to generate the system's output. The realization that at least two parties (for our purposes a human and a computer) are involved in a dialogue led Shneiderman to introduce the concept of **multiparty grammars** for the description of human-computer dialogues [39]. In a multiparty grammar, an augmented context-free grammar is used to describe the human input, machine response (i.e., acknowledgement or diagnostic) and some aspects of the interaction. Context-free grammars are augmented with three features:

[1] Labeling of nonterminals with an identifier distinguishing the party. For example <H: VALID-ACCT> is a nonterminal for a user input of a valid account number, and <C: ACCEPT-ACCT> is a nonterminal denoting the computer's response when accepting a valid user account.

[2] Assignment of currently parsed values to nonterminals, and correspondingly the use of square brackets to indicate that the value of the nonterminal is to be used in the generation of output. As an example, the following describes a simple dialogue between two humans meeting for the first time:

```
Reply::=        REPLY getid [resp: "Enter text field"  act: replybuf:=OPENFOREDIT_MSG(replyid)]
                        TEXT [act: SETTEXT_MSG($TEXT,replybuf)]
                        extras* [act: UPDATE_MSG(replyid,replybuf); CLOSEEDIT_MSG(replyid)]

getid::=        MSGID [act: replyid:=REPLY_MSG($MSGID)] | NULL [act: replyid:=REPLY_MSG(CurrentMsg)]

extras::=       extratos | extrraccs

extratos::=     TO toaddressee toaddressee*
toaddressee::=  ADDRESSEE [act: SETTO_MSG(replybuf,GETTO_MSG(replybuf)+$ADDRESSEE)]

extraccs::=     CC ccaddressee ccaddressee*
ccaddressee::=  ADDRESSEE [act: SETCC_MSG(replybuf,GETCC_MSG(replybuf)+$ADDRESSEE)]
```

BNF Specification of the "Reply" Command

Figure 2.3: Grammar Specification of a REPLY Command

```
<DIALOG>      ::=  <1: GREET>  <2: RESPOND>
<1: GREET>   ::=  GOOD MORNING MY NAME IS <1: NAME>
<1: NAME>    ::=  <1: IDENTIFIER>
<2: RESPOND> ::=  HELLO [<1: NAME>]
```

Note that the nonterminal [<1: NAME>] gets its value from the most recent parse, and is used to generate the response for party 2.

[3] A nonterminal which matches any string, if all other parses fail. This feature allows the designer to cope with incorrect syntactic forms.

An example of using the multiparty concept to describe a text-oriented logon procedure is illustrated in Figure 2.4.

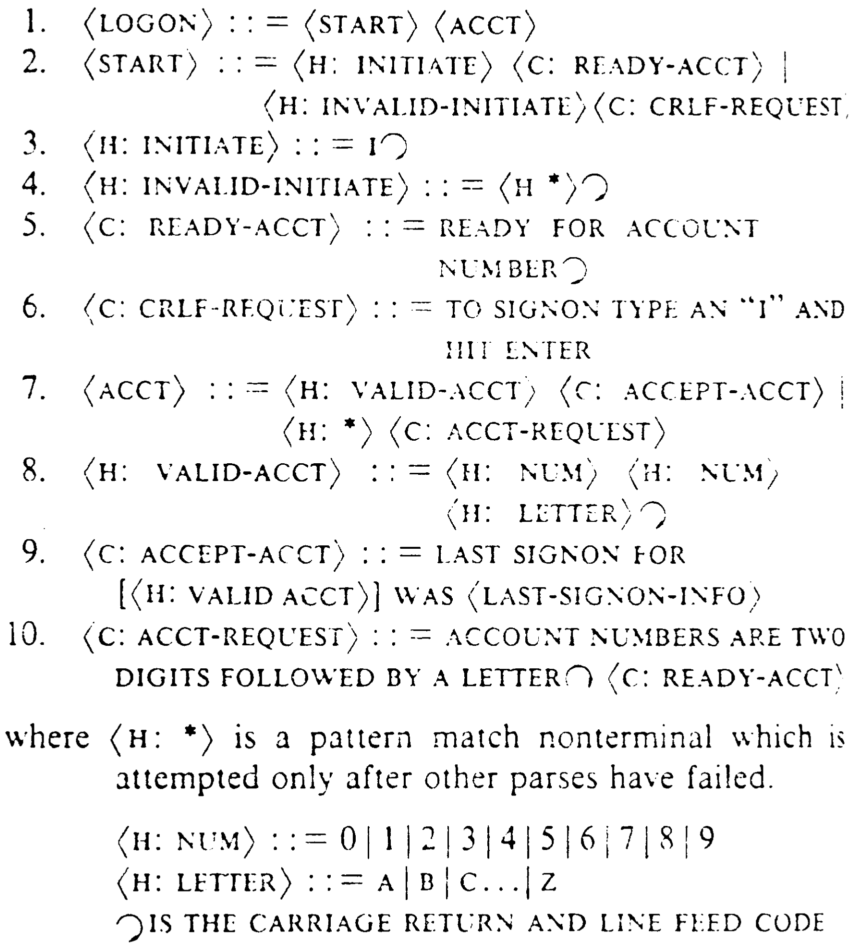
The discussion so far has been limited to describing text-oriented interfaces. A large percentage of current user interfaces are graphical in nature. To accommodate graphical features, Shneiderman [39] proposes the following additional tools to augment the multiparty grammars.

[1] Including special markers (as nonterminals) to describe features which merely change the visual presentation of characters, including underlining, reversal, blinking, font, typesize, intensity and color. For example, to support underlining, two markers are necessary: UNDERLINE and OFFUNDERLINE. The following is an example of underlining a portion of the system's output.

```
<C: RESPONSE> ::=  <C: FIRSTPART> <C: UNDERLINE>
                   <C: IMPORTANTPART> <C: OFFUNDERLINE>
```

[2] Including a special facility to describe the layout of a screen as specific windows. For example, the declaration of a screen layout which includes a status, workarea and command windows, is given below:

```
DECLARE SCREEN CONTAINING
  (WINDOW STATUS (1:4, 1:72) WITH STATUS-INFO
   WINDOW WORKSPACE (5:34, 1:72) WITH WORK-INFO
   WINDOW COMMANDS (35:40, 1:72) WITH COMMAND-INFO)
```



```
1.  ⟨LOGON⟩ : : = ⟨START⟩ ⟨ACCT⟩
2.  ⟨START⟩ : : = ⟨H: INITIATE⟩ ⟨C: READY-ACCT⟩ |
                  ⟨H: INVALID-INITIATE⟩⟨C: CRLF-REQUEST,
3.  ⟨H: INITIATE⟩ : : = I↩
4.  ⟨H: INVALID-INITIATE⟩ : : = ⟨H *⟩↩
5.  ⟨C: READY-ACCT⟩ : : = READY FOR ACCOUNT
                          NUMBER↩
6.  ⟨C: CRLF-REQUEST⟩ : : = TO SIGNON TYPE AN "I" AND
                          HIT ENTER
7.  ⟨ACCT⟩ : : = ⟨H: VALID-ACCT⟩ ⟨C: ACCEPT-ACCT⟩ |
                  ⟨H: *⟩ ⟨C: ACCT-REQUEST⟩
8.  ⟨H: VALID-ACCT⟩ : : = ⟨H: NUM⟩ ⟨H: NUM⟩
                          ⟨H: LETTER⟩↩
9.  ⟨C: ACCEPT-ACCT⟩ : : = LAST SIGNON FOR
       [⟨H: VALID ACCT⟩] WAS ⟨LAST-SIGNON-INFO⟩
10. ⟨C: ACCT-REQUEST⟩ : : = ACCOUNT NUMBERS ARE TWO
       DIGITS FOLLOWED BY A LETTER↩ ⟨C: READY-ACCT⟩
```

where ⟨H: *⟩ is a pattern match nonterminal which is attempted only after other parses have failed.

```
⟨H: NUM⟩ : : = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨H: LETTER⟩ : : = A | B | C . . . | Z
↩IS THE CARRIAGE RETURN AND LINE FEED CODE
```

Figure 2.4: Multiparty Grammar for a LOGIN Command

where each window is described in terms of rows and columns and the name of the symbol describing the contents of the window. Provisions should also be made for describing overlapping and nested windows, and for multiple screen layouts.

[3] Providing facilities to allow the information on the screen to be dynamically modified. The basis for this facility is the capacity to move a cursor to a specified position on the screen. Cursor movement can be described in various ways. For example, a cursor at position (3,1) could be moved to position (1,6) of the WORKINFO window by the following commands

[a] WORKINFO CURSOR TO (1,6)

[b] WORKINFO CURSOR UP 2 LEFT 5

[c] by hitting the keys { ↑,↓ <--, <--, <--, <--, <-- }

These three examples can be described by appropriate grammars as depicted in Figure 2.5. Note that facilities must also be provided to handle illegal cursor movement.

One of the major problems with associating computer actions/responses to output nonterminals is that the order in which the output nonterminal is performed is dependent on the parsing strategy. In a bottom-up parse, the action is performed after all the symbols on the right-hand side of the appropriate production have been recognized. In the case of a top-down parse, the action is performed as soon as the symbols before the output nonterminal in the appropriate production have been recognized. Figure 2.6 illustrates a grammar description of a dialogue for entering rubber band lines. Note that this specification is dependent on a top-down parsing strategy.

In order to alleviate some of the above problems, grammar productions that are involved with interaction can be enclosed in an **input-output tool** ([40, 41]) or **dialogue cell** ([42, 43]). Each interactive production is augmented with facilities to describe actions specific to interaction, including:

```
a) ⟨H: CURSOR MOVE⟩ : : =
     ⟨H: STARTING SYMBOL NAME⟩
     CURSOR TO (⟨H: INTEGER⟩, ⟨H: INTEGER⟩)

b) ⟨H: CURSOR MOVE⟩ : : =
     ⟨H: STARTING SYMBOL NAME⟩
     CURSOR ⟨H: VERTICAL⟩ ⟨H: HORIZONTAL⟩
   ⟨H: VERTICAL⟩ : : =
     UP ⟨H: INTEGER⟩ | DOWN ⟨H: INTEGER⟩
   ⟨H: HORIZONTAL⟩ : : =
     LEFT ⟨H: INTEGER⟩ | RIGHT ⟨H: INTEGER⟩

c) ⟨H: CURSOR MOVE⟩ : : =
     ⟨H: SINGLE MOVE⟩ |
     ⟨H: SINGLE MOVE⟩ ⟨H: CURSOR MOVE⟩
   ⟨H: SINGLE MOVE⟩ : : = ↑ | ↓ | → | ← .
```

Figure 2.5: Multiparty Grammar for Cursor Movement

```
line → button d1 end_point

end_point → move d2 end_point
          | button d3

d1 →
      { record first point }

d2 →
      { draw line to current position }

d3 →
      { record second point }
```

Figure 2.6: Grammar for Entering Rubber-band Lines

[a] prompting and initialization of local variables,

[b] the action to be performed at the end of the production,

[c] the echo produced for the interaction, and

[d] the value returned by the production.

It is important to note that most of the grammar-based UIMSs augment context-free grammars further to handle error recovery, undo processing, and control over the parsing strategy.

2.1.1.2. Finite-State Models

As early as 1969, Parnas [44], recognizing the inadequacy of the available tools, suggested the use of state-transition diagrams to define interactive user interfaces. A state transition diagram consists of states (represented by circles) and a set of transitions (represented by arrows) between states. Each transition is associated with an action; transitions correspond to user inputs and the associated action is performed by the system in response to the user input. One of the inherent problems with grammar-based models is that the concept of time sequence is implicit in the definition; that is, it is difficult to specify exactly **when** something occurs. State-transition diagrams, more commonly known as **transition networks,** override this problem as time sequences are explicitly defined.

Formally, a simple transition network (STN) is a 7-tuple, **M = (Q,S,P,D,R,Qo,F),** where

[1] **Q** is a finite set of states corresponding to the states in the diagram;

[2] **S** is a finite set of input symbols, which are the input tokens generated by the presentation component;

[3] **P** is a finite set of actions, which are the actions labeling the states of the transition diagram;

[4] **D** is a mapping from **Q x S** to **Q,** called the state transition function;

[5] **R** is a mapping from **Q** to **P,** called the action function;

[6] **Qo,** a member of **Q,** is the initial state of **M;** and

[7] **F,** a subset of **Q,** is the set of final states of **M.**

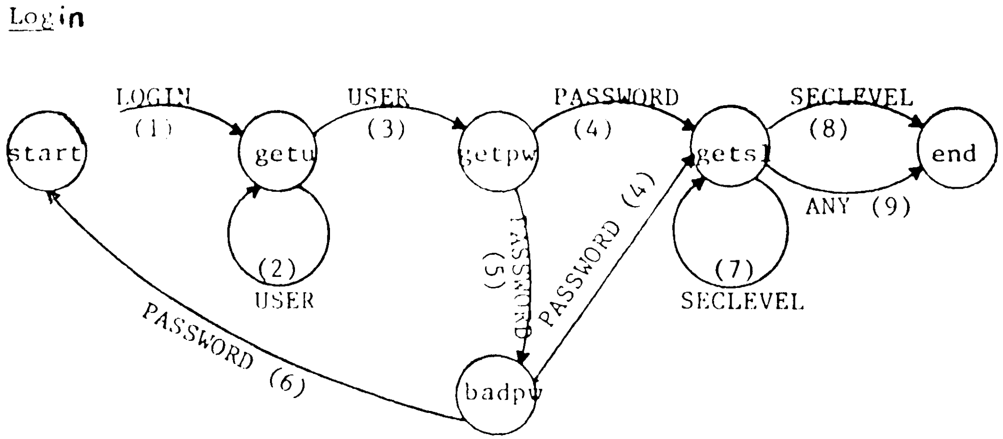
Note that this definition of a STN differs slightly from a finite-state machine in that the value of **D(q,a)** is the new state **q',** and the action **p,** a member of **P,** is given by **R(q').**

[38] provides examples of the use of state-transition diagrams to describe the "login" and "reply" commands that were presented in the previous subsection on grammars. Figures 2.7 and 2.8 depict the specifications of the "login" and "reply" commands respectively.

Action/responses generated by the computer can be attached to either the states or the transitions as shown in Figure 2.9, which depicts the specification of entering rubber band lines. Note that associating actions on arcs necessitates fewer states [45].

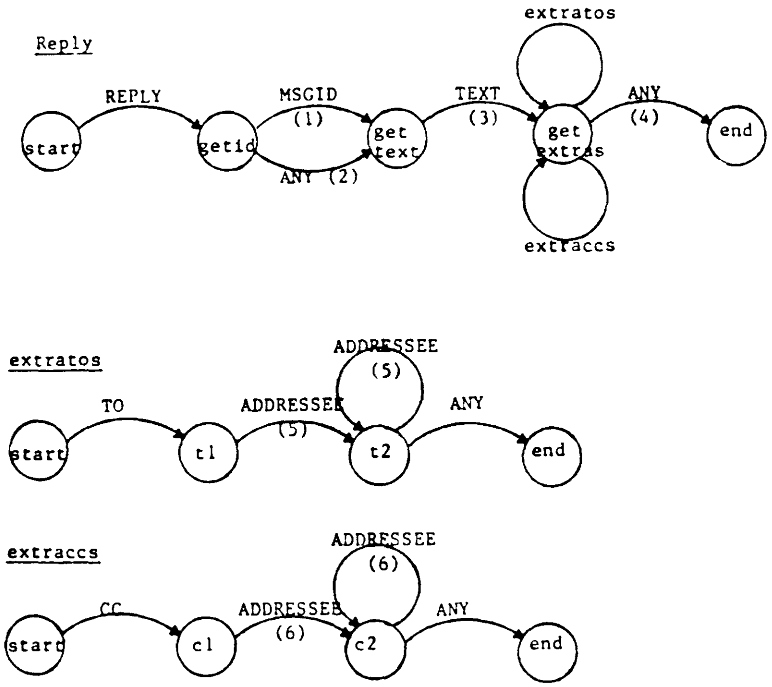
There are two problems with using simple transition networks to describe dialogues; the descriptions can get large, and the power of STNs limits the range of dialogues that can be described. One way of alleviating the first problem is to partition the network into subdiagrams, resulting in a main diagram and a number of subdiagrams. A subdiagram is a complete network that can be invoked from another diagram, akin to procedures in high-level programming languages. Note that the use of subdiagrams allow the specification to be divided into logical units, thereby making the network easier to understand and manage.

The power of transition networks can be increased in two ways. **Recursive Transition Networks** (RTNs) allow subdiagrams to call themselves recursively. **Augmented Transition Networks** (ATNs) augment RTNs with a set of registers that can hold arbitrary values, and a set of functions that can perform computations on the registers. Figure 2.10 uses RTNs to extend the rubber band example to describe entering polylines. A polyline is a sequence of rubber band lines, and the interaction allows for undoing (using backspace) the current (except the first) rubber band line. Note that it is not possible to describe entering polylines with STNs because



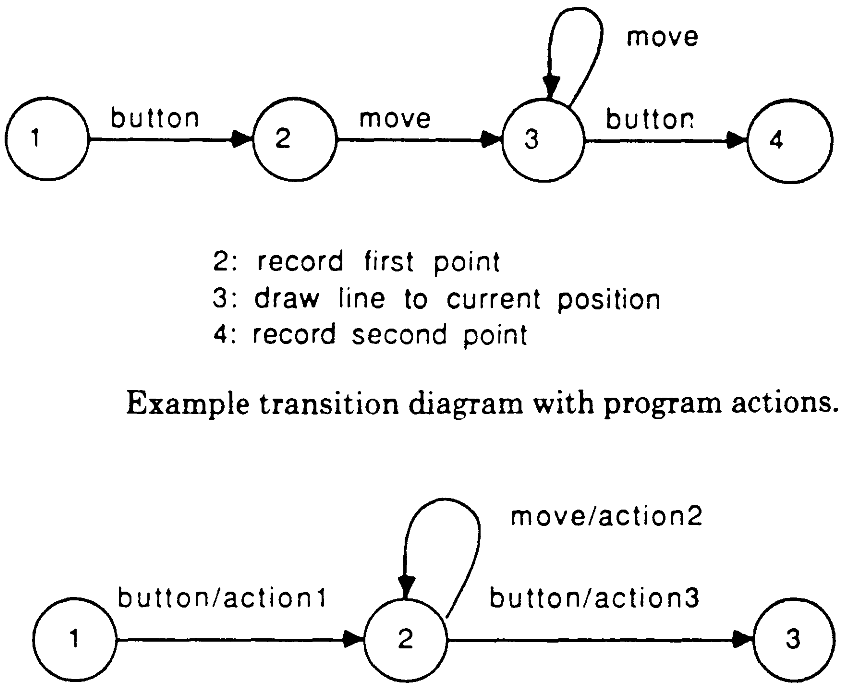
```
(1)   resp: "Enter name"
(2)   cond: not EXISTS_USER($USER)  resp: "Incorrect user name--reenter it"
(3)   cond: EXISTS_USER($USER)  resp: "Enter password"
(4)   cond: $PASSWORD=GETPASSWD_USER($USER)  resp: "Enter security level"
(5)   cond: $PASSWORD≠GETPASSWD_USER($USER)  resp: "Incorrect password--reenter it"
(6)   cond: $PASSWORD≠GETPASSWD_USER($USER)  resp: "Incorrect password--start again"
(7)   cond: $SECLEVEL>GETCLEARANCE_USER($USER)  resp: "Security level too high--reenter it"
(8)   cond: $SECLEVEL<=GETCLEARANCE_USER($USER)  act: CREATE_SESSION($USER,$PASSWORD,$SECLEVEL)
(9)   resp: "Your security level is Unclassified"  act: CREATE_SESSION($USER,$PASSWORD,Unclassified
```

Figure 2.7: State-Transition Diagram for LOGIN Command



```
resp: "Enter text field"  act: replyid:=REPLY_MSG($MSGID); replybuf:=OPENFOREDIT_MSG(replyid)
resp: "Enter text field"  act: replyid:=REPLY_MSG(CurrentMsg); replybuf:=OPENFOREDIT MSG(replyid
act: SETTEXT_MSG($TEXT,replybuf)
act: UPDATE_MSG(replyid,replybuf); CLOSEEDIT_MSG(replyid)
act: SETTO_MSG(replybuf,GETTO_MSG(replybuf)+$ADDRESSEE)
act: SETCC_MSG(replybuf,GETCC_MSG(replybuf)+$ADDRESSEE)
```

Figure 2.8: State-Transition Diagram for REPLY Command



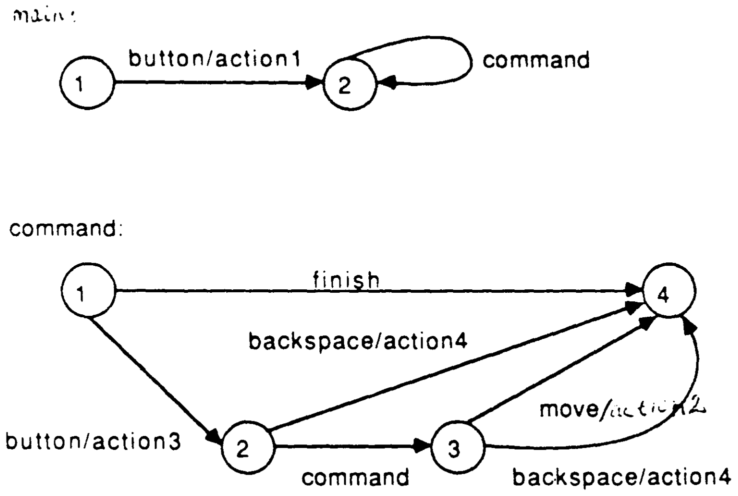
2: record first point
3: draw line to current position
4: record second point

Example transition diagram with program actions.

action1: record first point
action2: draw line to current position
action3: record second point

Example transition diagram with actions on arcs.

Figure 2.9: Transition Diagram with Actions and Responses



action1: record first point
action2: draw line to current position
action3: record next point
action4: erase last point

Figure 2.10: Transition Diagram for Polyline Example

there is no mechanism for undoing lines. Figure 2.11 uses ATNs to describe the same polyline dialogue without the use of subdiagrams. To handle undoing lines, a register is used to record the number of lines entered and a function is used to check whether the number of lines entered is greater than one. Note that this ATN description also includes a cancel feature which terminates the dialogue with an empty polyline. RTNs cannot handle the cancel feature due to the nesting of subdiagram calls.

Shaw [46] lists the following as reasons for the inadequacy of both grammar-based formalisms and state-transition diagrams for defining interactive graphics systems.

[1] The characterization of command languages as **strings** of symbols ignores the different techniques that are possible for the input sequence.

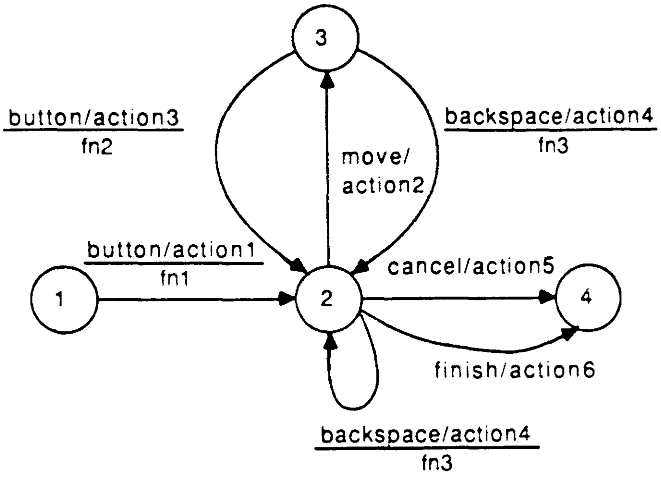
[2] In an interactive graphical system, several different and independent graphical objects (each with their own command language) may be active at any given time. It is not sufficient to describe the language of each object independently, as there is generally some interaction between the languages.

[3] It is frequently desirable to have several partially-specified command sequences active at the same time. String-oriented formalisms normally impose a strict ordering on command elements.

[4] Most methods cannot effectively describe the interactions between user commands and the system that interprets these commands.

The language of **flow expressions** [47, 48] is proposed to overcome the above difficulties. They describe the flow of system entities, such as resources, messages, jobs, commands and control, through sequential and concurrent software components such as programs, procedures, processes and modules. As an example, a line drawing command sequence can be described as

$$(\text{Line } (x\ y\ U\ r\ d))^{*}\ @$$



action1: record first point
action2: draw line to current position
action3: record next point
action4: erase last point
action5: erase polyline
action6: return polyline

```
fn1:  count:=1; return(true);
fn2:  count:=count+1; return(true);
fn3:  if count = 1 then
           return(false);
      else
           count := count-1;
           return(true);
```

Figure 2.11: Polyline Dialogue with Cancel

where Line, x, y, r, d, and @ are atomic elements. Thus, a line is represented by zero or more repetitions of the Line command followed by either an (x,y) pair or an (r,d) pair, and terminated by @. Figure 2.12 shows a finite state machine for recognizing the command.

A more interesting example is one which describes the interactions between an interactive user, H, at a terminal and the underlying system. A user issues a command followed by a carriage-return (CR) and then waits for a system response (RES). The system is cyclic, allowing a new user to start a session when the terminal is free. The flows through both subsystems are depicted both as expressions and diagrams in Figure 2.13. In the figure, Wi and Ri represent semaphores and are interpreted as "wait on signal i", and "send signal i" respectively. As shown in the Figure, the T/H waits and signals ensure that both subsystems are properly synchronized at logoff. The other waits and signals are used to describe the handshaking protocols between the two subsystems.

Finally, it is important to note that state-transition diagrams have also been used to describe non-trivial interactive systems, as demonstrated in [49, 50].

2.1.1.3. Event Models

The event dialogue model, developed by Green [51], is based on the idea of generating and handling events. Events are generated by input devices, and within the dialogue component by tokens sent from other components of the user interface, or from events that need to invoke/create/destroy other events. Generated events are sent to one or more currently existing event handlers that can handle the type of the event generated. The behavior of an event handler is described by a template similar to a procedure in high-level programming languages. Each template describes parameters, local variables, the event types that can be processed, the actual processing, and initialization code. Event handlers are created by specifying the template, and providing values for the parameters.

Formally, an event is a 3-tuple, **E = (i,m,d),** where

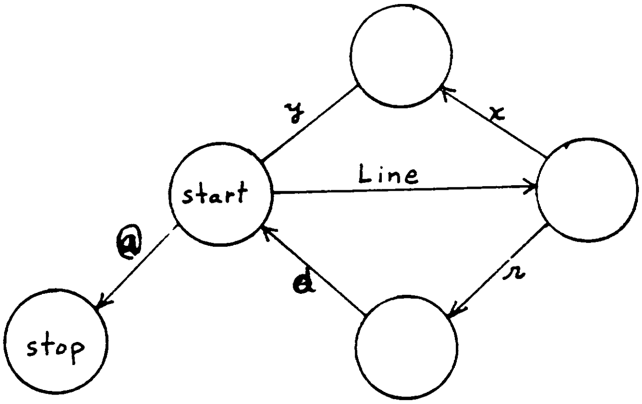


Figure 2.12: Finite State Machine for Drawing Lines

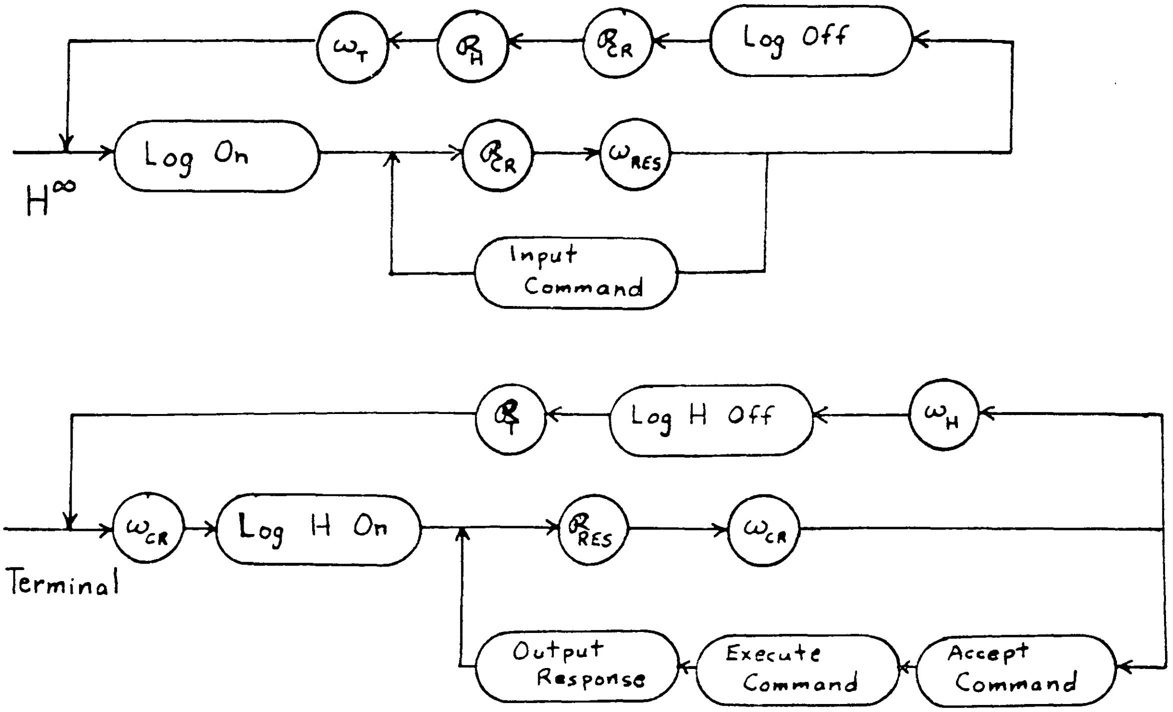


Figure 2.13: Flow Expressions for Multi-user System

[1] **l** is a member of **I,** where **I** is a finite set of symbols that are used as names for the event handlers in the event system;

[2] **m** is a member of **M,** where **M** is a finite set of symbols that are used as event names; and

[3] **d** is a member of **D,** where **D** is a domain (possibly infinite) that is used for event values.

Event handlers are procedures written in a high-level programming language.

Figures 2.14 and 2.15 depict the specification for entering rubber lines and polylines (with cancel) respectively, using events.

The major power of using events is in the description of multi-threaded dialogues. Most of the interactive systems available today are window-based, allowing for concurrent dialogues. A good example of multi-threaded dialogues is a cut-and-paste operation in a window-based multi-file editor. A description of such a dialogue must allow for a section of a file in one window to be cut and pasted to another file in a different window. This involves communication between the dialogues running in the two windows, and can be simulated by sending an event from one window to the other. Note that the other dialogue models cannot describe such multi-threaded dialogues.

2.1.1.4. Discussion on Dialogue Models

This section is concerned with the descriptive power and usability of the three dialogue models that have been presented, in an attempt to determine their relative strengths and weaknesses.

Intuitively, the event model is most powerful as it can describe multi-threaded interfaces (such as the cut-and-paste example) that the others cannot. Note that transition networks are equivalent in power to deterministic push-down automata. The only difference between the two is the action function associated with transition networks. It is also important to note that the context-free grammars employed within UIMSs are usually restricted to a subset (**deterministic**

```
EVENT HANDLER line;

 TOKEN
  button Button;
  move Move;

 VAR
  int state;
  point first, last;

 EVENT Button  DO {
  IF state == 0 THEN
    first = current position;
    state = 1;
  ELSE
    last = current position;
    deactivate(self);
  ENDIF;
 };

 EVENT Move DO {
  IF state == 1 THEN
    draw line from first to current position;
  ENDIF;
 };

 INIT
  state = 0;

END EVENT HANDLER line;
```

Figure 2.14: Event Handler for the Rubber-band Line Example

```
EVENT HANDLER polyline_cancel;

 TOKEN
  button Button;
  move Move;
  backspace Backspace;
  cancel Cancel;
  finish Finish;

 VAR
  point_count : integer;
  point_list : list of point;
  int state;

 EVENT Button DO {
  IF state = 0 THEN
   point_list = current position;
   state = 1;
   point_count = 1;
  ELSE
   add current position to point_list;
   point_count = point_count + 1;
  ENDIF;
 };

 EVENT Move DO {
  IF state = 1 THEN
   draw line from last position to current position;
  ENDIF;
 };

 EVENT Finish DO {
  return(point_list);
  deactivate(self);
 };

 EVENT Backspace DO {
  IF point_count > 1 THEN
   remove last point from point_list;
   point_count = point_count - 1;
  ELSE
   output "can't delete first point";
  ENDIF;
 };

 EVENT cancel DO {
  return(empty_list);
  deactivate(self);
 };

 INIT
  state = 0;

END EVENT HANDLER polyline_cancel;
```

Figure 2.15: Event Handler for Polyline with Cancel

context-free grammars) that can be handled by popular parsing techniques (usually LL(1) or LALR(1)). It has been shown that the set of languages generated by deterministic context-free grammars and deterministic push-down automata are similar [45]. Practically, the event model has the power of the programming language that it is embedded in, giving it the descriptive power of a Turing machine. Furthermore, Green [52] has devised algorithms to convert descriptions in the transition network and context-free grammar models to the event model. This then shows that the event model is at least as powerful in describing dialogues as the other two models.

As concerns usability, it is very difficult to identify characteristics that make a particular model usable. Usability of a dialogue model is dependent on the dialogue designer and the application. However, the conversion routines presented by Green makes it possible for UIMSs to support all three models.

In conclusion, it is appropriate to develop UIMSs based on the event model as it allows dialogues to be specified in any of the three dialogue models.

2.1.2. User Interface Management Systems

A User Interface Management System (UIMS) provides a user interface designer with the facilities to specify, generate, validate, manage and evaluate user interfaces. The common characteristic of all UIMSs is the specification mechanism that is provided to describe all or part of the intended user interface. This specification mechanism is based predominantly on one of the three dialogue models presented in the previous subsections.

The first UIMS that was built was Newman's Reaction Handler [53], which provided facilities to specify user interfaces through an interactive state-transition diagram editor. Since then, numerous UIMSs have been developed, both in commercial and educational environments. The UIMSs that have been built are categorized below, by reference, according to the specification mechanism that is used.

[1] Grammar-based UIMSs: [41], [43], [54], [55], and [56, 57, 58].

[2] Finite-state based UIMSs: [38], [44], [53], [59], [60], [61], [62], [63], [64], [65], [66], [67] and [68].

[3] Event based UIMSs: [69], [70], [71], [72] and [73].

The following subsections present a representative UIMS from each category.

2.1.2.1. The SYNGRAPH System

The SYNGRAPH (Syntax directed GRAPHics) system [56, 57, 58] is a user interface generator which has been developed as part of the Automated Human Interfaces (AHI) project being carried out at Arizona State University. This research applies the principles of syntax analysis, parser generation and data abstraction to automatically generate user interfaces. Figure 2.16 depicts the basic architecture of interactive systems, which forms the model around which the research has been developed. The Interaction Specification forms the core of the system. It defines the valid commands in terms of their syntax and the legal operands and also describes the command's representation (i.e., their names or symbols) and the devices or techniques that are to be used. The interaction module is automatically generated from this specification.

The specification of the user interface is made up of a **lexical specification** and a **syntactic specification.** The lexical specification describes the binding of input devices and interaction techniques to logical token symbols, which in turn are used in the syntactic specification. There are seven primitive input devices supported by SYNGRAPH: menu items, locators, valuators, function buttons, keys, characters and picks. Associated with each input device are the following properties: the type of the value that is returned, whether an event or sampled device, the action to be performed when it is enabled, the prompt for its selection, the acknowledgement when it is selected, and the effect the device has on transitions. These properties are predefined and managed automatically by SYNGRAPH. An example of a lexical specification is given in

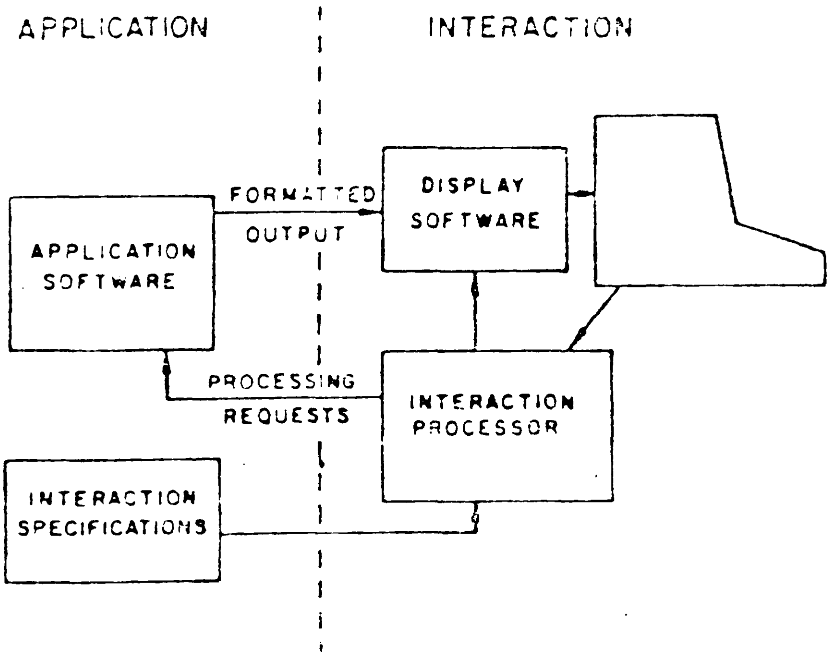


Figure 2.16: SYNGRAPH System Architecture

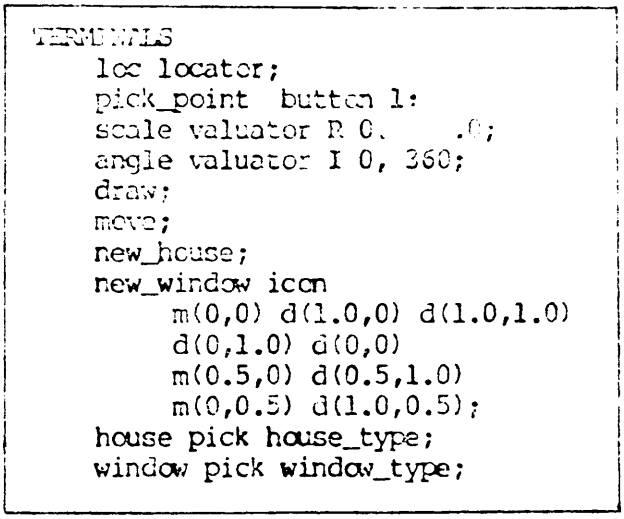
Figure 2.17. Here, LOC refers to the locator; PICK_POINT to the stylus tip switch; SCALE and ANGLE are real and integer valuators respectively; DRAW, MOVE and NEW_HOUSE are all menu items; and NEW_WINDOW is an iconic menu item (in the Figure "m" stands for a move command, and "d" for a draw command). Finally, the HOUSE and WINDOW tokens specify that a picture of the appropriate type is to be picked.

The syntactic specification describes the prompts, echoes and sequencing of the tokens specified in the lexical specification which constitute valid inputs. The syntactic specification is expressed using a modified BNF notation, allowing for optional phrases, repeating phrases (between braces), as well as alteration (using "|"). Syntax processing also controls the management of display and input resources. The user interface is described in terms of sub-dialogues within which resources are allocated. Thus the specification mechanism allows the designation of nonterminals as defining a new level of interaction. When a new level is entered, all of the tokens used within that level (directly or indirectly) become enabled. Thus, a level defines a specific configuration of virtual input devices. An example of a syntactic specification is depicted in Figure 2.18(a), and the corresponding menu sets for each level are shown in Figure 2.18(b). To handle interaction semantics, the specification mechanism allows each nonterminal to have a semantic routine associated with it. The semantic specification is described in terms of a Pascal parameter string (the semantic function) and a set of Pascal declarations (the semantic attributes).

The specification is parsed to generate the user interface. Prompting, menu and device management, and error detection and handling are also automatically generated. Finally, SYNGRAPH provides a CANCEL operation which allows the user to jump to a previous state in a single step.

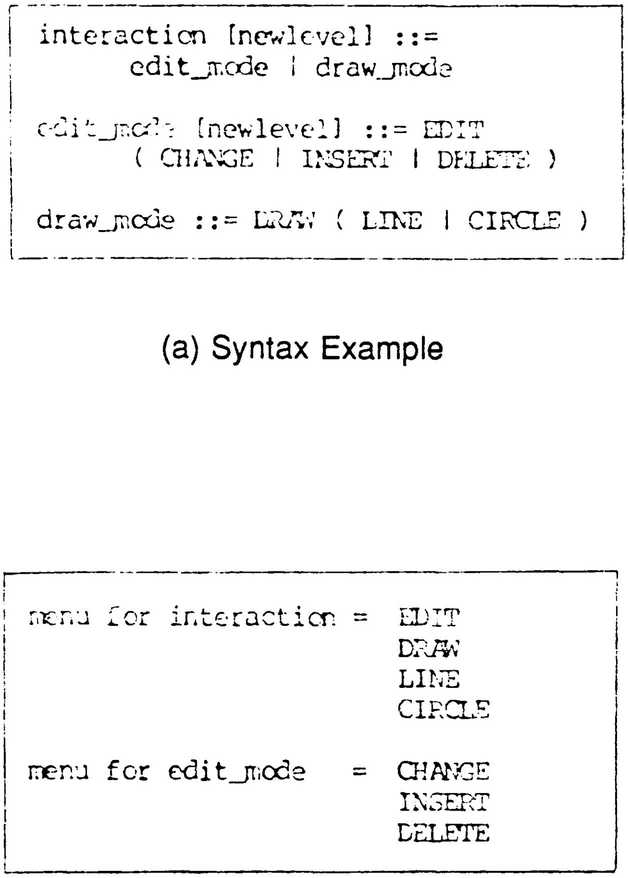
2.1.2.2. The Abstract Interaction Handler

The AIH (Abstract Interaction Handler) system [63] is the user interface management system of the Information Display Systems project, being developed at the George Washington



```
TERMINALS
     loc locator;
     pick_point  button 1:
     scale valuator R 0.      .0;
     angle valuator I 0, 360;
     draw;
     move;
     new_house;
     new_window icon
          m(0,0) d(1.0,0) d(1.0,1.0)
          d(0,1.0) d(0,0)
          m(0.5,0) d(0.5,1.0)
          m(0,0.5) d(1.0,0.5);
     house pick house_type;
     window pick window_type;
```

Figure 2.17: Lexical Specification in SYNGRAPH



```
interaction [newlevel] ::=
      edit_mode | draw_mode

edit_mode [newlevel] ::= EDIT
      ( CHANGE | INSERT | DELETE )

draw_mode ::= DRAW ( LINE | CIRCLE )
```

(a) Syntax Example

```
menu for interaction =  EDIT
                        DRAW
                        LINE
                        CIRCLE

menu for edit_mode   =  CHANGE
                        INSERT
                        DELETE
```

(b) Menu Sets for Levels

Figure 2.18: Syntactic Specification in SYNGRAPH

University. The aim of this project is to apply the principles of top-down design and functional abstraction to the design of interactive graphical systems.

The AIH environment is made up of several components:

[1] an interaction language (IL), based on augmented transition networks, which allows the syntax of interactive dialogues to be described,

[2] an interpreter for the interaction language (ILI), which activates appropriate application functions, passing to them the inputs acquired from the user.

[3] a set of style modules which describe style-dependent attributes such as level of prompting,

[4] a library of user profiles which contain information on personalized styles of interaction,

[5] a screen handler which handles the output to different windows,

[6] a library of interaction techniques which describe how devices are used for user input, and

[7] an underlying device-independent graphics package based on the CORE standard [74], which provides the graphics support.

A block diagram of the overall system configuration is given in Figure 2.19.

The interaction language specifies the user actions, the sequencing of user actions and the semantic module to respond to user actions. The language presents seven task types: select-category, select-operation, select-entity, position, quantify-integer, quantify-real and text, which are elaborations of the interaction tasks developed in [14]. An interactive system is specified as a network of nodes, each node being one of the seven types above. The definition of an interaction node is presented in Figure 2.20(a). Figures 2.20(b) and (c) describe the node records and basic types needed in describing an interaction node.

An example of the specification of a style module is presented in Figure 2.21. Style modules can invoke various interaction techniques to accomplish the types of task in the interac-

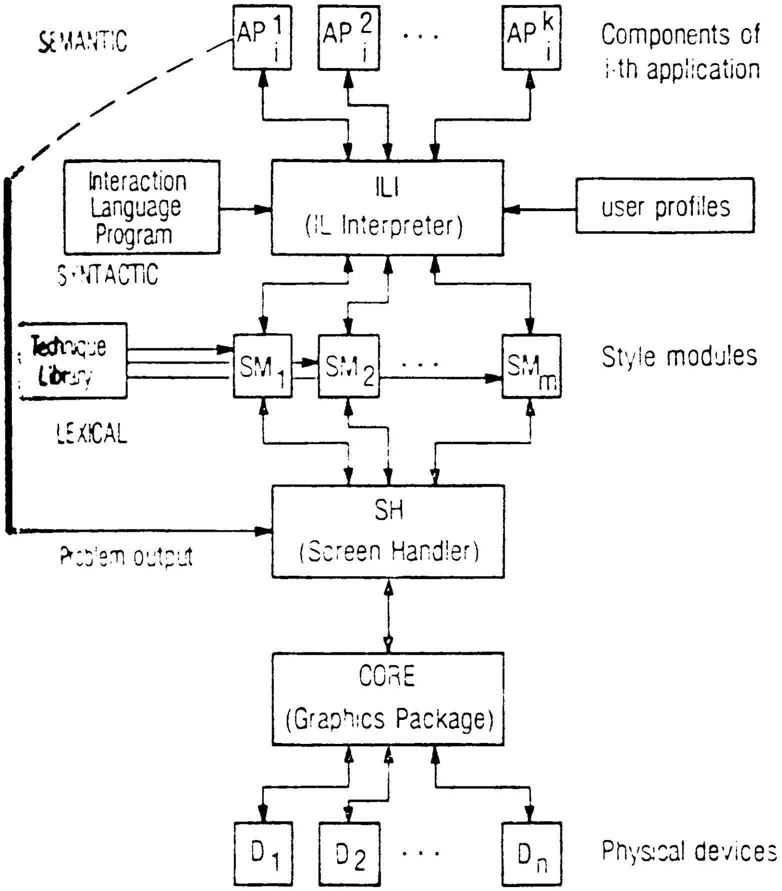


Figure 2.19: AIH System Configuration

```
type
interaction__node is
    record
        who__am__i:   node__id;
        what__kind:   task__type;
        prompt:       prompt__help__triple;
        --[now variable stuff depending on interaction type]
        case what__kind of
            when select__category   => set of category__records;
            when select__operation  => set of operation__records;
            when select__entity     => set of entity__records;
            when position           => position__record;
            when quantify__integer  => integer__record;
            when quantify__real     => real__record;
            when text               => text__record;
        end case;
    end record;
```

(a) Interaction Node

```
type
task__type is (
    select__category, select__operation,
    select__entity, position,
    quantify__integer, quantify__real, text);
prompt__help__triple is
    record brief: text__string;
           full:  text__string;
           help:  text__string;
    end record;
action__item is
    record invokes: semantic__module__id;
           next:    node__id;
    end record;
```

(b) Node Record

Figure 2.20: Specifying an Interactive System in AIH

```
type
category_records is
    record  which_one:      keyword;
            description:    display_info;
            what_to_do:     action_items;
    end record;
operation_records is
    record  which_one:      keyword;
            description:    display_info;
            what_to_do:     action_items;
    end record;
entity_records is
    record  which_one:      identifier;
            description:    display_info;
            what_to_do:     action_items;
    end record;
position_record is
    record  x_range:        real_range;
            y_range:        real_range;
            what_to_do:     action_items;
    end record;
integer_record is
    record  how_big:        integer_range;
            what_to_do:     action_items;
    end record;
real_record is
    record  how_big:        real_range;
            what_to_do:     action_items;
    end record;
text_record is
    record  how_big:        max_string_length;
            what_to_do:     action_items;
    end record;
```

(c) Basic Types in AIH

Figure 2.20: Specifying an Interactive System in AIH

```
Display  Level_10
    Task_code:       in Task_Name;
    Ret_response:   out Response_Description);
type Response_Description is
    record
        case task_type of
            when select_category.
                 select_operation,
                 select_entity         =>     internal_token;
            when position              =>     locator_pair;
            when quantify_integer      =>     integer_value;
            when quantify_real         =>     real_value;
            when text                  =>     text_record
        end case
    end record;
```

Figure 2.21: A Style Module in AIH

tion language. Figure 2.22 illustrates the available interaction techniques. The AIH enforces a set of predefined windows: prompt, response, help, error, results and concurrent. These windows are implemented and handled by the screen handler package. All the information needed by the AIH system is organized into a data base. Access to the information is through predefined routines.

The description of the application specified by the interactive language is interpreted by the interactive language interpreter; that is, the interpreter traverses the network defined by the interaction language. The interpreter accepts the user input, determines the currently active interaction task, communicates the relevant information to the interaction techniques which do the actual interacting with the user. The style modules and the user profiles influence the selection of the interaction techniques.

2.1.2.3. The University of Alberta UIMS

The University of Alberta UIMS (UAUIMS), [71], is an attempt at evaluating the feasibility of using the Seeheim model (as presented in Figure 2.1) as the basis for UIMSs. The UAUIMS therefore provides tools to describe all three components of the user interface, that is, the presentation, dialogue control and application interface components. The UAUIMS is based on the event model, but provides facilities to the designer to describe the dialogue in the other two models as well. These grammar-based and finite-state based descriptions are translated automatically into an event based description.

The UAUIMS is divided into two major components, the user interface design component and the run-time support component. The user interface design component provides the necessary tools to help the designer specifying the presentation, dialogue control and application interface components of the user interface. The run-time support environment converts the user interface specification into a complete executable user interface, and provides facilities to support the execution of the user interface.

TASK TYPE	TECHNIQUE NAME	INPUT DEVICE	FEEDBACK
Select_Category	Select name of category from a menu.	Kevboard	(i) Characters are echoed at appropriate location as they are typed in. (ii) The selected category blinks in displayed menu.
	Select ordinal number of category in menu	Keyboard	(i) Ordinal number is echoed at appropriate location. (ii) The selected category blinks in displayed menu
	Pick category from a menu.	Tablet	The selected category blinks in the displayed menu.
Select_Operation	Select name of operation from from a menu	Keyboard	(i) Characters are echoed at appropriate location as they are typed in. (ii) The selected operation blinks in displayed menu.
	Select ordinal number of operation in menu	Keyboard	(i) Ordinal number is echoed at appropriate location (ii) The selected operation blinks in displayed menu.
	Pick operation from a menu.	Tablet	The selected operation blinks in the displayed menu.
Select_Entity	Select name of entity from a menu.	Keyboard	(i) Characters are echoed at appropriate location as they are typed in. (ii) The selected entity blinks in displayed menu.
	Select ordinal number of entity in menu.	Keyboard	(i) Ordinal number is echoed at appropriate location. (ii) The selected entity blinks in displayed menu.
	Pick entity from a menu.	Tablet	The selected entity blinks in the displayed menu.
Position	Provide a pair of real values.	Keyboaru* *There is some "low level" interaction involved in using a Keyboard to enter a position	The values are echoed at the appropriate location on display device.
	Provide a locator pair.	Tablet	(i) The cursor blinks at the selected position. (ii) The coordinates 8f the selected position are echoed
Quantify_Integer	Provide an integer.	Keyboard	The value provided is echoed at appropriate location on display device.
Quantify_Real	Provide a real .	Keyboard	The value provided is echoed at appropriate location on display device.
	Provide a valuator.	Valuator* *Underlying software simulates this using a tablet and a display screen	The selected value is echoed by simulation software at corresponding position on displayed scale.
Text	Provide a test string	Keyboard	Text string is echoed at appropriate location as it is typed in

Figure 2.22: Interaction Techniques Supported by AIH

The design of the presentation component is supported by an interactive layout program, called IPCS (Interactive Presentation Component Specification), that allows a designer to interactively design the screen layout, interaction techniques and display techniques for a particular user interface. Support for IPCS is provided by an underlying window system, WINDLIB, which implements the basic graphics primitives and window management routines.

The design of the dialogue control component is supported by a set of tools, one for each of the dialogue models supported by UAUIMS. The event model is supported by a programming language based on C. This language provides the facilities to define appropriate event handlers. Figure 2.23 presents the structure of an event handler. The finite-state model is supported by an interactive graphical transition diagram editor that is used to create and edit recursive transition networks. The UAUIMS provides no support for the grammar-based model in its current version, but plans to support it in future versions.

The support for the application interface component is limited to a definition of the mapping between user interface tokens and their associated functions within the application. Future versions of UAUIMS will extend this limited support.

The specifications from each component of the user interface design are converted to a common base format, known as the Event Based Internal Format (EBIF). An assembler converts the user interface specification into a file of C routines and appropriate tables used by the run-time support component. The C routines are then compiled by a standard compiler to form an executable user interface. Figure 2.24 depicts the process of converting an event language program into a user interface.

2.2. SIMD Framework: User Interface Construction Kits

The major objective of UIDEs within the SIMD framework is to provide the system designer with facilities/tools to rapidly construct user interfaces for any application. Most SIMD UIDEs allow for the initial construction of standardized consistent user interfaces to facilitate rapid development. Additional facilities are then provided to refine or fine-tune these user

```
Eventhandler event_handler_name Is

 Token
  token_name event_name ;
            .
            .
            .

 Var
  type variable_name = initial_value ;
            .
            .
            .

 Event event_name : type {
    statements
 }
           .
           .
           .
 Event event_name : type {
    statements
 }

end event_handler_name;
```

Figure 2.23: Structure of an Event Handler

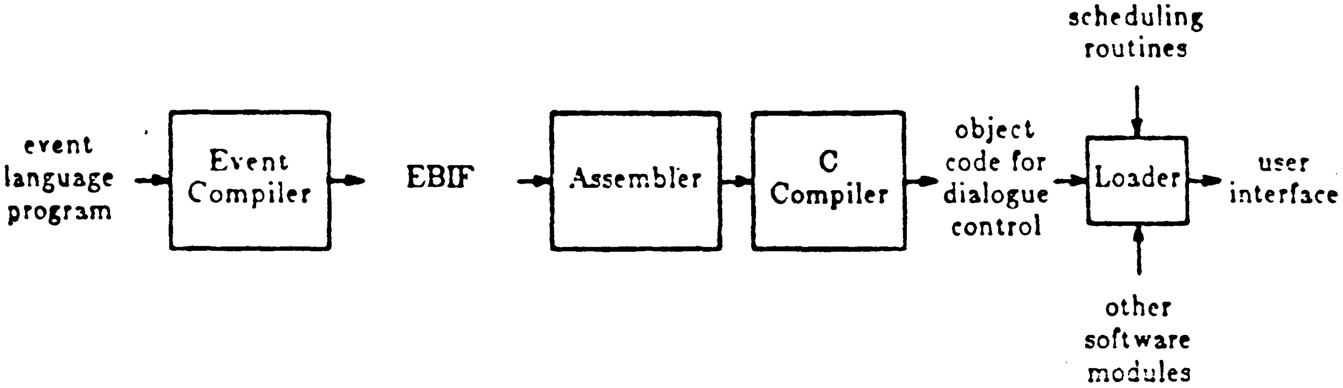


Figure 2.24: Converting an Event Program into an Interface

interfaces to better suit the peculiarities of the particular application.

We can identify the following four different types of construction kits depending on the ease in which user interfaces can be developed, that is, the sophistication of the user interface development environment.

[1] **Graphics Packages:** Graphics packages provide the facilities to manipulate the basic elements of graphical input and output. Graphical input is concerned with managing and manipulating input devices such as keyboards, mice, tablets, etc. Graphical output concerns the display of color, text and geometric shapes (such as lines, circles, rectangles, etc.). Note that conventional programming languages, such as Pascal, C, and Ada, depend entirely on graphical packages to develop the user interface component of interactive systems.

[2] **Window Systems:** Window systems extend graphics packages to provide facilities to both create and manage interaction of one or more applications. The base window system, a graphics package with some additional high-level facilities (such as menus), is the substrate on which sophisticated multi-application interactive systems are built. Window systems provide two additional components to manage interaction. The window manager provides facilities to manipulate overlapping windows. The input manager controls the interface between input devices and applications, that is, which applications get input from which devices. Note that the development of sophisticated interactive systems, such as software development environments, rely on window systems for building and managing interaction.

[3] **User Interface Frameworks:** A framework provides facilities to construct the components of an abstract design for a particular application. For example, a compiler framework provides facilities to construct a lexical analyzer, a parser, a symbol table, a type checker, and a code generator. Frameworks must also be concerned with defining and managing the interfaces between the components. User interface frameworks are concerned with the

construction of the following major components: the presentation of application information to the user, the acceptance of user input, and the invocation of application functions. Furthermore, user interface frameworks must be concerned with defining and managing the interface between the application and the user interface. Frameworks are akin to libraries that support conventional programming environments.

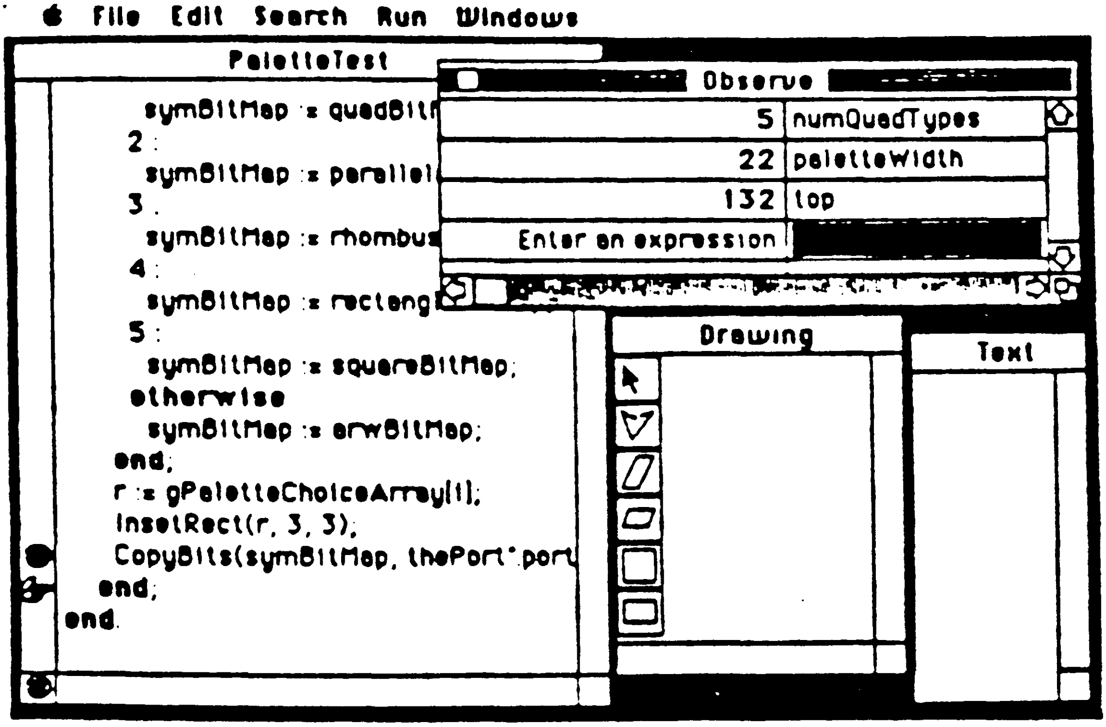
[4] **User Interface Toolkits:** A user interface toolkit is a collection of high level tools that provide facilities to conFigure and construct user interfaces. Ideally, toolkits are built on top of frameworks and allow the designer to interact with the underlying framework to construct new applications.

Since this survey is concerned primarily with user interface development environments, the following discussion will focus exclusively on environments that support user interface frameworks and/or toolkits. These environments are usually built, however, on a underlying graphics package or window system. It is important to note that research on graphics packages and window systems have developed to a point where standards are available. CORE [74] and GKS [75] are well established standards for graphics packages, while the X window system [76] is rapidly gaining popularity and acceptance as a standard for window systems.

The following subsections present two popular user interface frameworks and two representative user interface toolkits.

2.2.1. The MacApp Framework

The MacApp framework [77] provides Apple Macintosh system designers, a prefabricated standard user interface for any application. The standard Macintosh user interface is depicted in Figure 2.25. The basic premise of MacApp is to provide a user interface that automatically handles characteristics that are common to all applications (such as resizing of windows), and to allow designers to plug in details specific to the application (such as the contents of each window). Since the user interface code is provided by MacApp, designers have the flexibility of fine tuning the interface to suit the peculiarities of the particular application.



```
    symBitMap := quadBitM
   2 :
    symBitMap := parallel
   3 :
    symBitMap := rhombus
   4 :
    symBitMap := rectang
   5 :
    symBitMap := squareBitMap;
   otherwise
    symBitMap := arwBitMap;
  end;
  r := gPaletteChoiceArray[i];
  InsetRect(r, 3, 3);
  CopyBits(symBitMap, thePort^.port
 end;
end.
```

5	numQuadTypes
22	paletteWidth
132	top
Enter an expression	

Figure 2.25: A Macintosh User Interface Example

An abstract MacApp application consists of one or more windows, one or more documents, and a single application object. A window is associated with each document and is concerned with displaying the state of the document. Each window consists of a set of views, where each view is concerned with displaying a part of the associated document. A new interactive system necessitates only the definition of the documents and the view objects that render images of the documents. Figure 2.26 presents the objects (and their relationships) for a typical Macintosh application.

Specifically, the MacApp framework consists of the following six components.

[1] The **application** component is the main controller of the interactive system. Its main purpose is to receive (input) events, classify the events, and invoke appropriate application-specific event handlers. It also provides a set of standard global functions such as opening documents and quitting the system. This component is also in charge of creating document objects.

[2] The **document** component provides the hooks for describing the application. Documents manage and manipulate the application data structures, and are responsible for communicating their current state to view objects. Documents are also responsible with reading and writing their data from and to backing store.

[3] The **window** component manages standard Macintosh windows and provides standard facilities for moving, resizing and scrolling.

[4] The **frame** component is used to subdivide a window into independent parts. All Macintosh windows consists of at least two frames, a fixed pallette area and a scrollable drawing area. Frames can also be scrolled.

[5] The **view** component defines the image to be drawn within a frame. This is the view of the application state within the appropriate document.

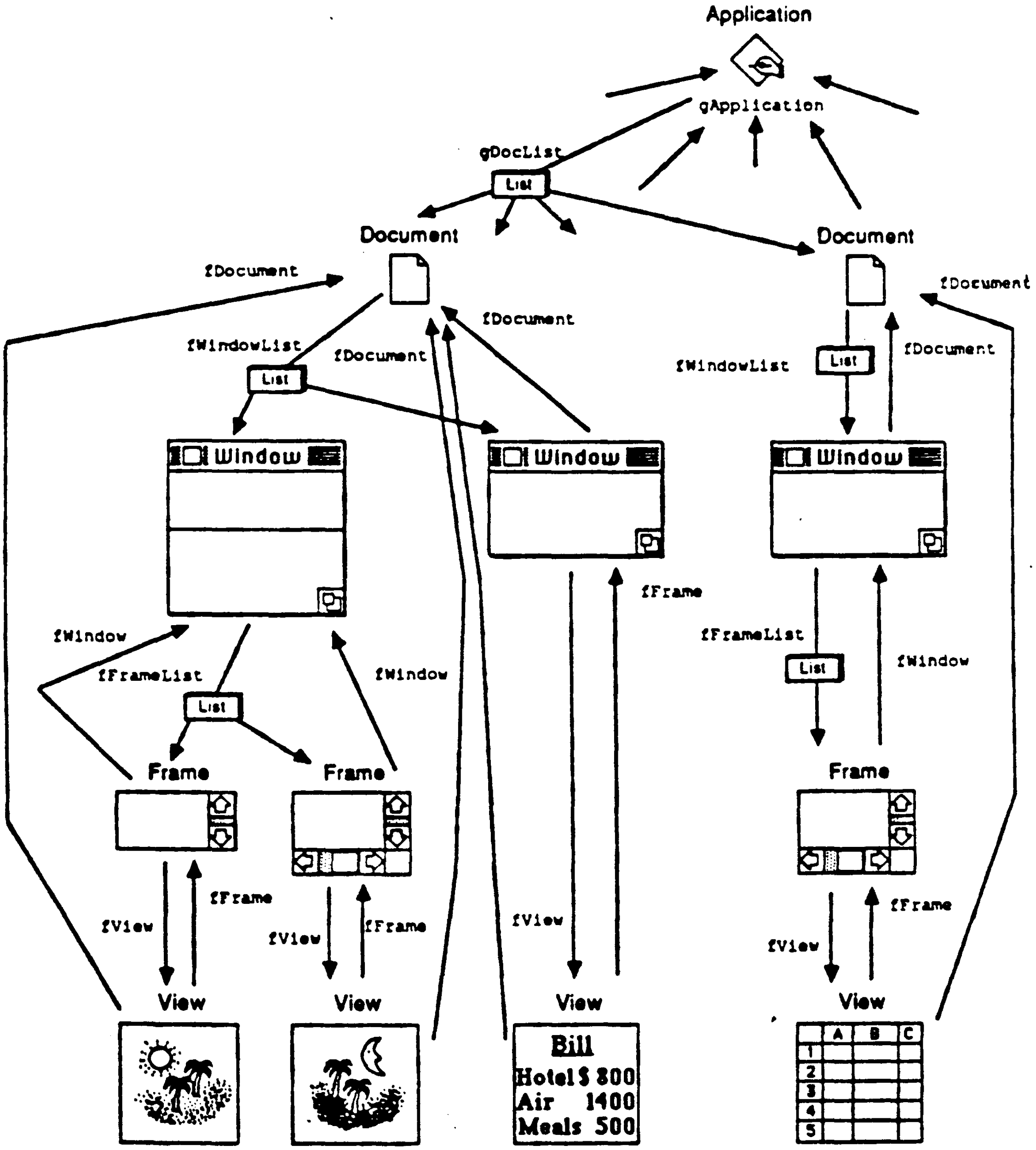


Figure 2.26: The Objects in a typical MacApp Application

[6] The **command** component is used to interface between the application and the user interface. Commands are created by documents, are responsible for performing the command (that is, changing application data and updating views), and most importantly for undoing the command, if necessary.

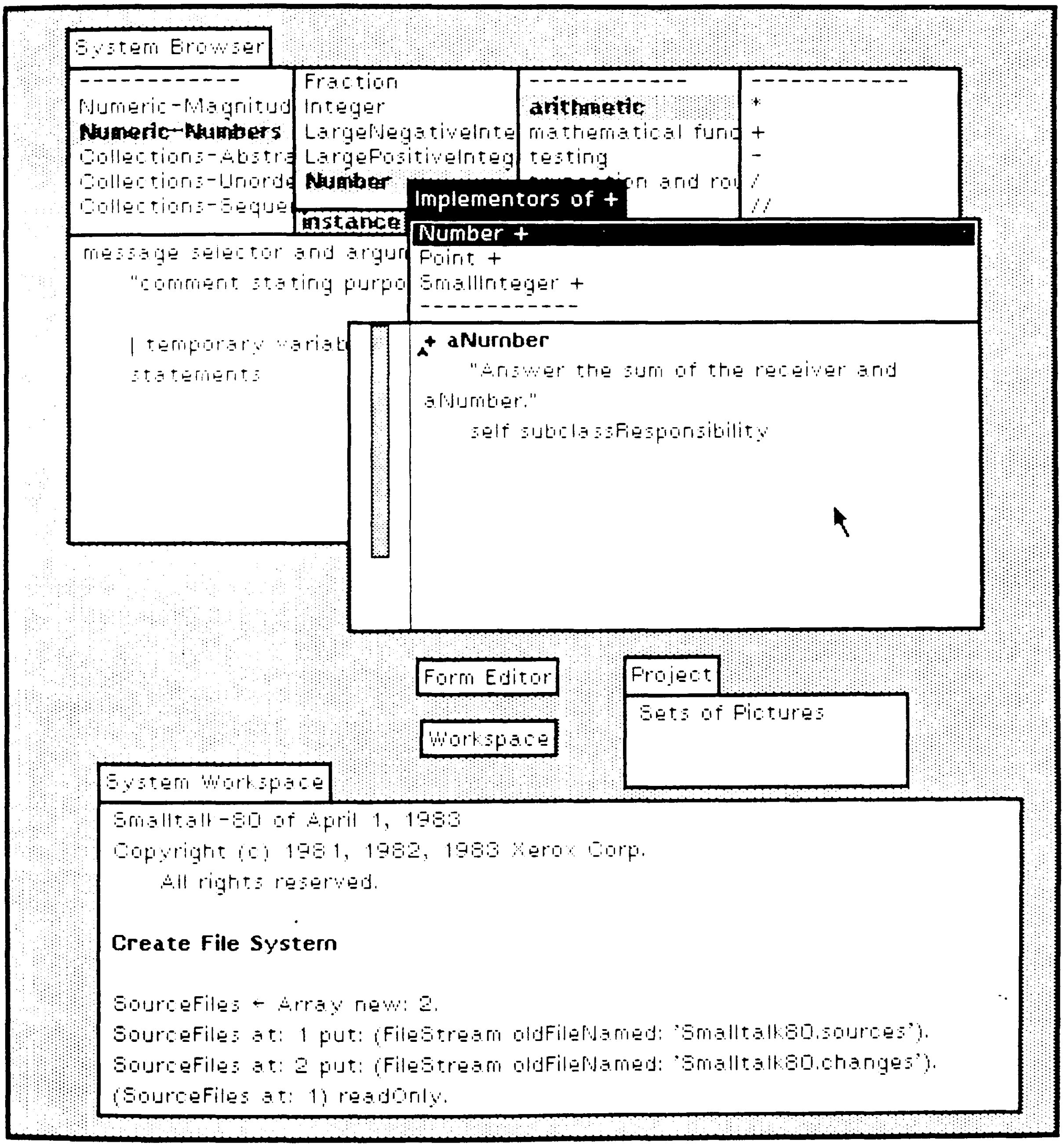
2.2.2. Smalltalk's Model-View-Controller Framework

The Model-View-Controller (MVC) framework [78] facilitates the construction of standardized user interfaces for the Smalltalk environment [79]. A typical Smalltalk user interface is depicted in Figure 2.27. As with the MacApp framework presented above, the MVC framework constructs a user interface with standard facilities common to all applications, and provides the necessary hooks to allow application-specific behavior to be attached. Once again, the availability of the code for the user interface allows designers to fine tune the interface to suit the particular application.

An abstract Smalltalk application consists of a model, which in turn consists of a set of related view-controller pairs. Views are used to display the state of the application while controllers manage interaction with the user. Controllers act as the interface between the user and the application, as well as the interface between the application and its views. The interaction between the three components is shown in Figure 2.28. The standard interaction cycle is as follows. The active controller accepts user input and invokes the appropriate application function within the model. The model performs the requested action, and broadcasts the change to its dependents (that is, its set of view-controller pairs). Views update their displays through application (model) provided information.

Specifically, the MVC framework consists of the following components.

[1] The **model** component provides the facilities to define an application. It also facilitates the connection of view-controller pairs as dependents and the ability to broadcast changes to these dependents.



```
+ aNumber
     "Answer the sum of the receiver and
aNumber."
     self subclassResponsibility
```

```
Smalltalk-80 of April 1, 1983
Copyright (c) 1981, 1982, 1983 Xerox Corp.
    All rights reserved.

Create File System

SourceFiles ← Array new: 2.
SourceFiles at: 1 put: (FileStream oldFileNamed: 'Smalltalk80.sources').
SourceFiles at: 2 put: (FileStream oldFileNamed: 'Smalltalk80.changes').
(SourceFiles at: 1) readOnly.
```

Figure 2.27: A Typical Smalltalk User Interface

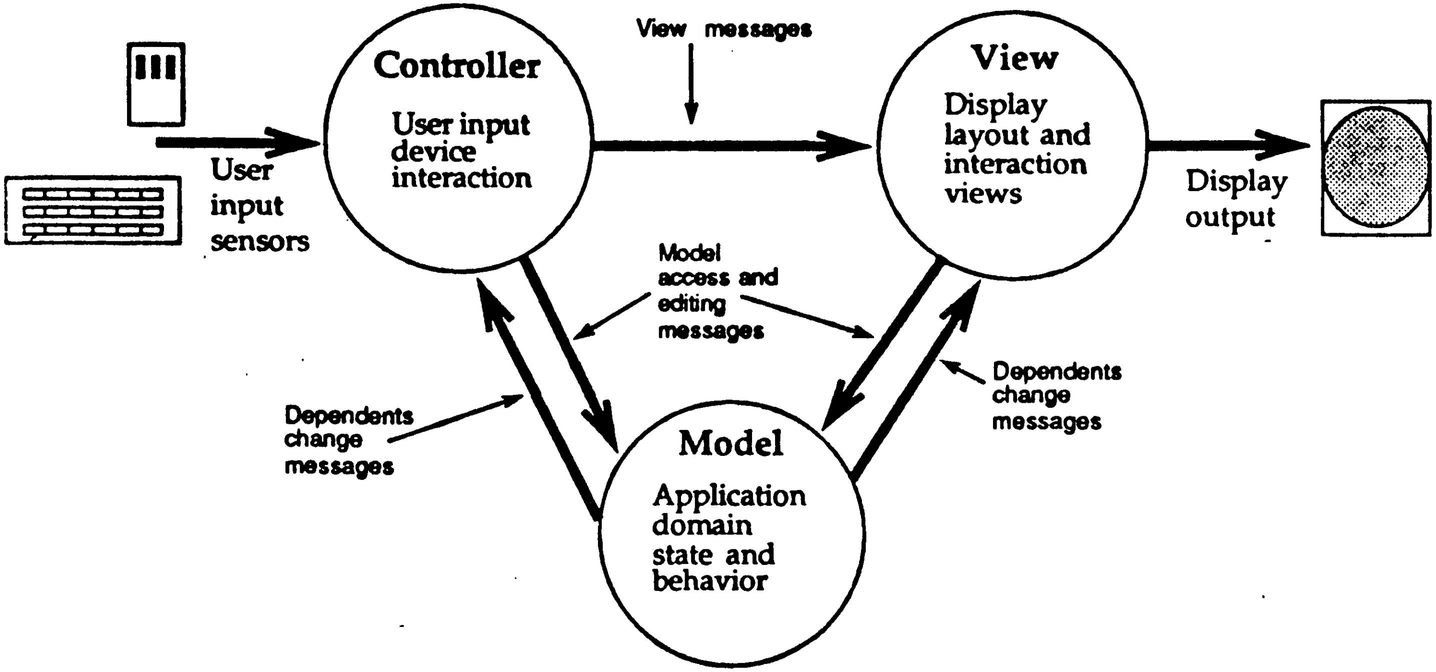

View messages
Controller
User input device interaction
View
Display layout and interaction views
User input sensors
Display output
Model access and editing messages
Dependents change messages
Dependents change messages
Model
Application domain state and behavior

Figure 2.28: Model-View-Controller Relationships

[2] The **view** component is concerned with display. It is equivalent to windows in most systems, allowing for the creation and management of subwindows (or subviews). Views provide standard facilities for clipping, transformation and display. The top level view is a standard system view that provides standard facilities for resizing, movement and scrolling. The actual display of application specific information is provided by the model component. Note that each view has exactly one model and one controller that it is associated with.

[3] The **controller** component provides the basic facilities to coordinate the user with the appropriate view and application model. Standard controllers are provided to handle the standard system view, a menu controller for mouse-based pop-up menus, and a scroll controller to handle scroll bars for views. The active controller is in charge of obtaining user input and invoking the appropriate application specific function to handle the input.

2.2.3. The Flair System

The FLAIR (Functional Language Articulated Interactive Resource) system [80] developed at TRW allows a designer to rapidly prototype a system's user interface, independent of the application. Thus, the main aim is to allow designers to rapidly realize and manipulate conceptualized dialogues to test various approaches to the design of the user interface. This approach allows the end users to be involved early in the design process to define the "best" user interface that will be implemented in the real system.

The FLAIR system, also called the Dialog Design Language (DDL), is a comprehensive software tool that allows system designers to describe human-computer interactions, as well as human factors researchers to evaluate the man-machine interface. DDL comprises the following tools.

[1] An ACM Core standard graphics package which provides the basic primitives to manipulate graphics objects. These primitives include drawing of lines, boxes, circles; writing of text in different sizes and fonts; color selection; area filling; pan and zoom of the screen; reading (writing) pixel images to (from) the screen; erasure of a single item or an entire

display; and cursor controlled free hand drawing.

[2] A relational data base management system, INGRES, which manages system and user-defined data relationships. User defined data can be associated with a particular graphics symbol (on the screen) and can be queried at any time.

[3] A workstation that provides a mix of the latest technology in input/output devices, as illustrated in Figure 2.29, ranging from voice to high resolution color graphics monitors.

The overall functionality of the DDL is shown in Figure 2.30. The DDL allows the designer to interactively build either a static frame or more importantly to create command menu hierarchies for dynamic scenarios. The system is voice menu-driven, i.e., the list of operations that FLAIR can perform in any given context are provided as menus, and these operations are invoked by voice input. The commands of the DDL consists of 15 root words which in turn can activate 85 commands to accomplish a task, as depicted in Figure 2.31. The screen layout is divided into six windows located on two workstations, as illustrated in Figure 2.32. The computational component in Figure 2.30 performs the various functions provided by the DDL. The resulting output operations can be the display of objects on specified windows, requests for more information from the user, or synthesized voice-output to the user. A designer uses the FLAIR system to design the user interface, which can be saved, edited and played back.

2.2.4. The MIKE System

The Menu Interaction Kontrol Environment (MIKE) system [81] was developed at the Brigham Young University Interactive Software Systems Laboratory. MIKE is a toolkit for rapidly prototyping a default textual user interface, and for refining the default user interface into a highly graphical sophisticated user interface to suit the particular application.

A MIKE user interface specification is made up of the following three components:

[1] the functions that define the application,

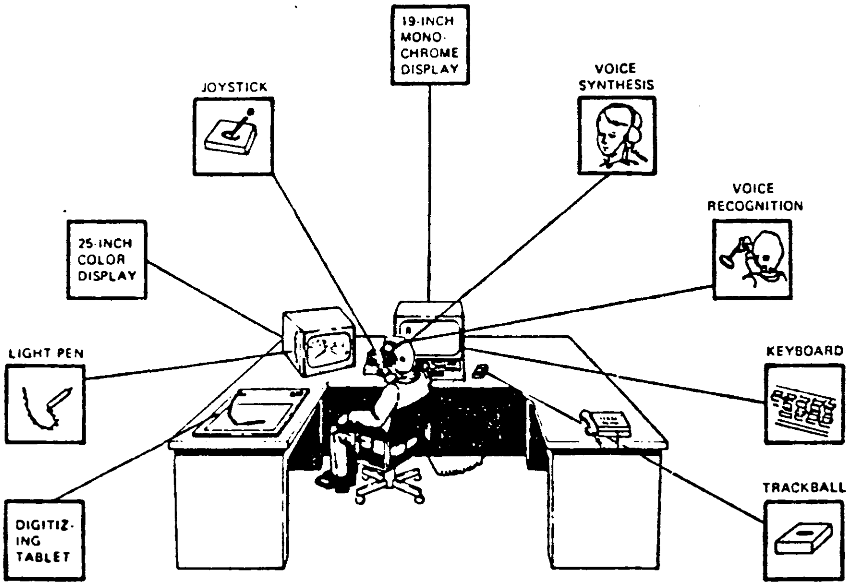


Figure 2.29: A FLAIR Work Station

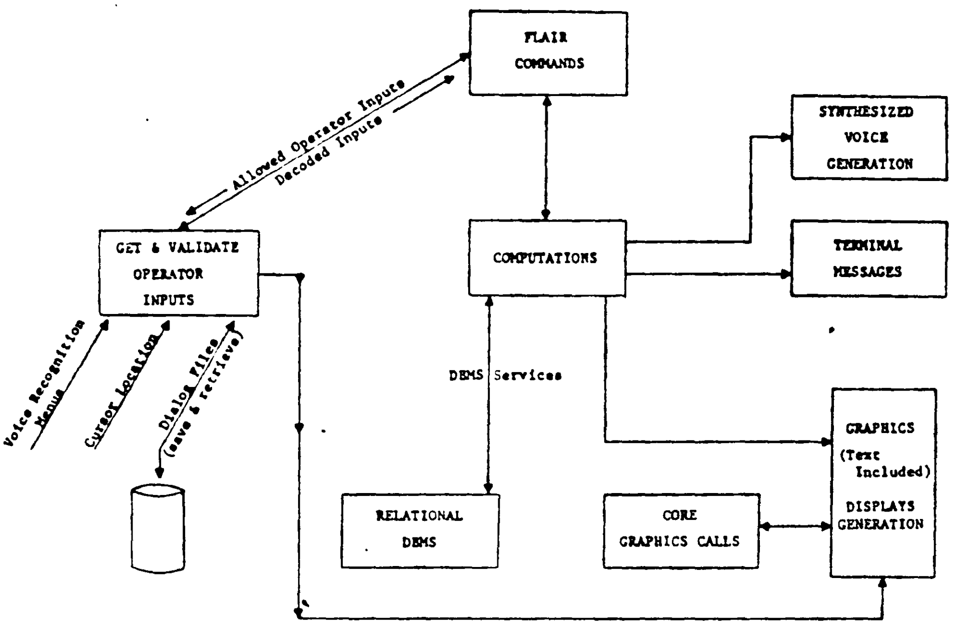


Figure 2.30: Functional Flow for the FLAIR DDL

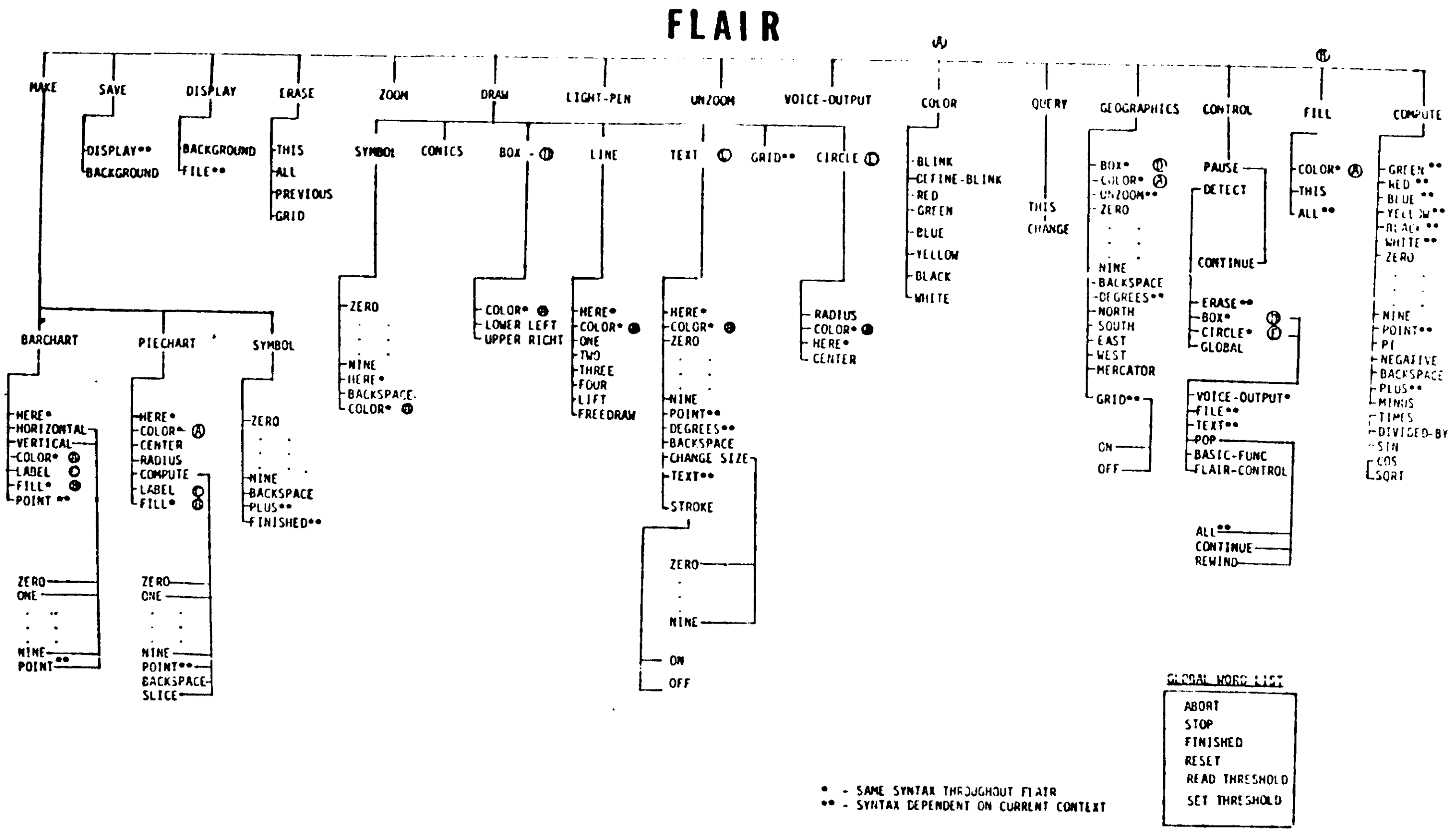


Figure 2.31: FLAIR DDL Commands

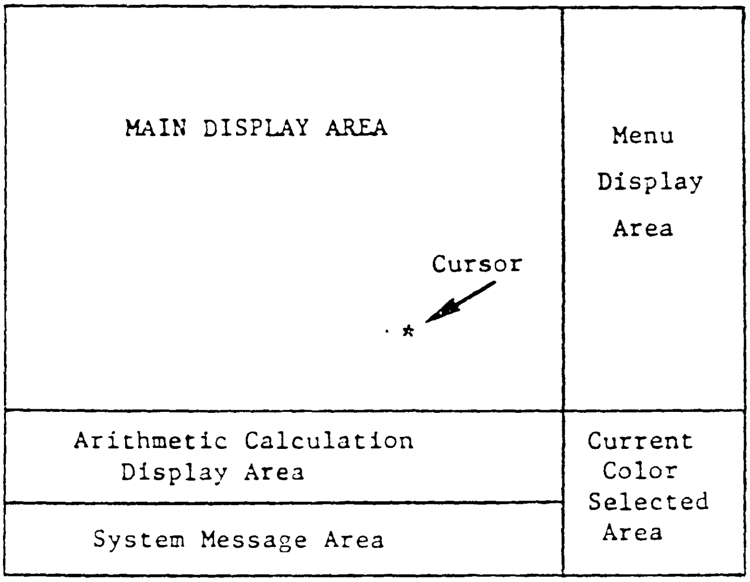

MAIN DISPLAY AREA
Menu Display Area
Cursor
*
Arithmetic Calculation Display Area
Current Color Selected Area
System Message Area

Figure 2.32: FLAIR Window Areas

[2] bindings between application functions and interaction techniques, and

[3] viewport definitions to describe screen layout and visual presentation.

All of the above components are specified using interactive tools.

The initial prototype definition for the user interface consists of a definition of a set of types relevant to the application, and a corresponding set of functions that can operate on objects of each type. For example, a user interface for a simple circuit layout program is concerned with the following types: Resistor, Capacitor, Wire and Connection. Figure 2.33 shows a list of possible functions that can be defined for these types. This initial prototype definition is supported by an interactive interface editor. With this limited amount of information, MIKE generates a default user interface through the process shown in Figure 2.34. The interface editor generates an interface profile, which contains the description of the user interface. The interface editor also generates Pascal code that defines the interface between the user interface and the application. This generated code is then compiled and linked with the application-specific code and MIKE's standard user interface code to produce the interactive program.

After the default user interface has been generated, MIKE provides facilities to refine the presentation component in the interface profile. MIKE allows editing of three aspects of the presentation component. The first concerns menus and the mapping of input events to application specific functions. Menus can be divided into hierarchies; menu items can have external names or icons associated with it; function keys can be associated with menu manipulation, and special keys can be associated to handle global inputs such as rubout, cancel and quit.

The second aspect of the presentation component that can be edited concerns prompts, echoes and help. The interface editor allows the designer to associate arbitrary prompt strings with every parameter that necessitates user input. Echoes can be defined to provide visual or textual feedback to the user, and finally help strings can be associated with every object of the user interface.

```
CreateResistor(Ohms: Integer; C1 : Connection; C2 : Connection)
CreateCapacitor(Farads: Integer; C1 : Connection; C2 : Connection)
CreateWire(C1 : Connection; C2 : Connection)

PickConnection(Where : Point) : Connection
  If Where is over an existing connection, then that connection is returned.
  Otherwise a new connection is generated at Where
PickResistor(Where : Point) : Resistor
PickCapacitor(Where: Point) : Capacitor
PickWire(Where: Point) : Wire

DeleteResistor(R : Resistor)
DeleteCapacitor(C : Capacitor)
DeleteWire(W : Wire)

MoveResistor(R : Resistor; To : Point)
MoveCapacitor(C : Capacitor; To : Point)
MoveWire(W : Wire; To : Point)

ChangeResistance(Of: Resistor; Ohms: Integer)
ChangeCapacitance(Of: Capacitor; Farads: Integer)


ResistanceOf(R : Resistor) : Integer;
CapacitanceOf(C : Capacitor) : Integer;

SaveCircuit(FileName: InString)
DiscardCircuit
LoadCircuit(FileName: InString)
```

Figure 2.33: Types for a Circuit Layout Program

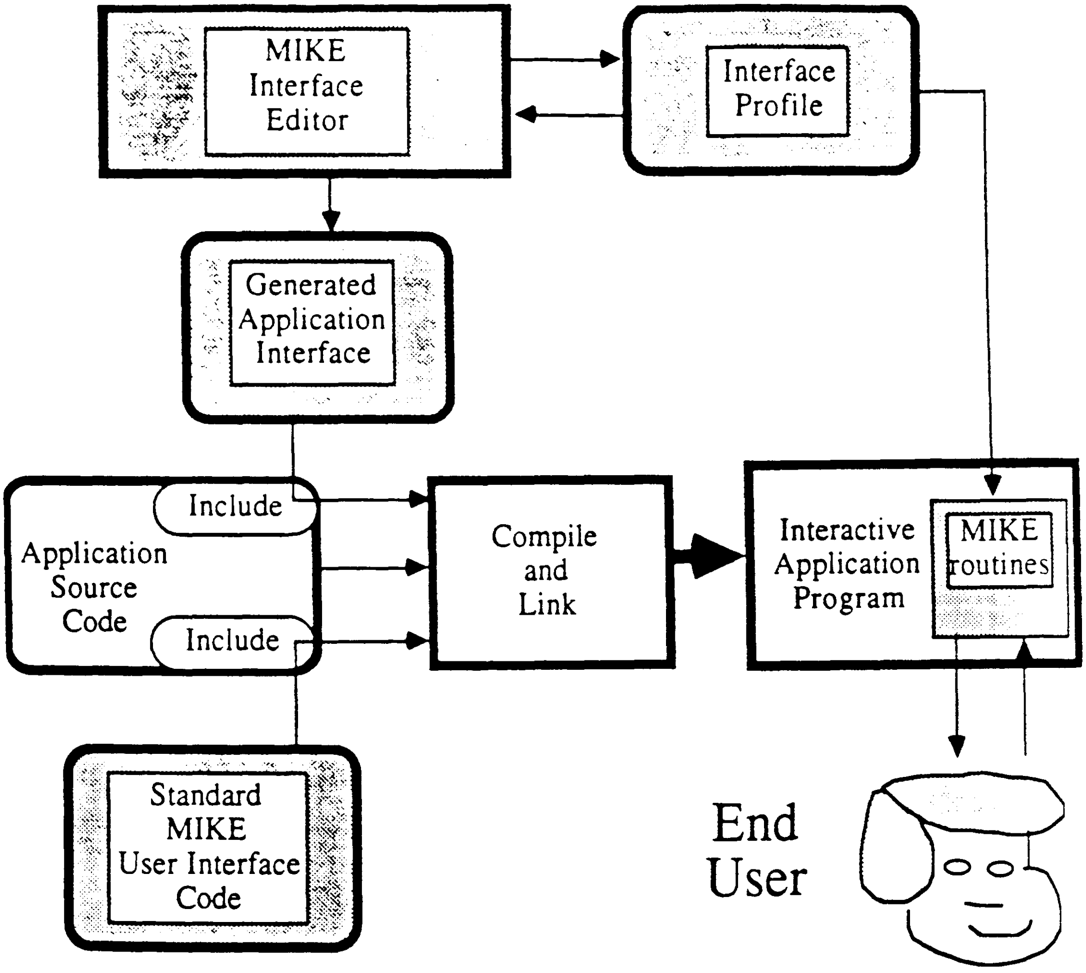


Figure 2.34: Generating a User Interface in MIKE

The final aspect of the presentation component that can be edited is the layout of the screen and the facilities for directly manipulating displayed objects. These facilities allow the designer to create a graphical user interface.

2.3. MISD Framework: End-User Customizable Interfaces

The main emphasis of UIDEs within the MISD framework is to provide facilities to end users to define personalized user interfaces for a single application. The major disadvantage of the MISD approach is cost, since it is necessary to develop user interface generators for every application.

There has been only one implementation of a MISD UIDE in the literature. This system is described in the following subsection.

2.3.1. The GIGL System

GIGL (Generator of Interactive Graphical Languages) is a prototype system to generate graphical interfaces, developed by Bournique as part of his doctoral research at the University of Pittsburgh [82]. The basic premise is that a universal "virtual" interaction language can be developed for a particular application area. This predetermined virtual interaction language describes all the possible aspects of graphical interfaces that can be used for the application, enumerating all the alternatives that are possible. For instance, the universal language can describe a menu response as

```
<menu response> ::= "chosen menu item is illuminated"|
                    "chosen menu item is blinked" |
                    "chosen menu item changes color"
```

which indicates that visual feedback is provided by either brightening, flashing or changing the color of the chosen menu item.

Once this universal language has been defined, the generation of personalized graphical interfaces is straightforward: choosing amongst a set of alternatives; or whether a feature is provided or not; and the specification of some needed parameters (e.g., window size).

Context-free grammars are used to describe the universal interaction language. Terminals of the language describe system actions. Figure 2.35 depicts a (portion of a) universal interaction language developed for graphical interactions. GIGL itself is a syntax-directed translator of the universal interaction language. The language is compiled internally into a library of semantic routines. GIGL is an interpreter of these routines.

The functional structure of GIGL is depicted in Figure 2.36. GIGL operates in two modes: **Interrogate** and **Interact.** The interrogate mode is used to generate a specific user interface. GIGL interactively presents queries from each syntactic class (or grammar production) defined in the universal interaction language. Some of these queries are parametric, i.e., a value must be specified. An example query is presented below

```
SPECIFY PARAMETERS OF WORK AREA:
LOWER LEFT COORDINATES?  0 100
UPPER RIGHT COORDINATES? 800 1000
```

Queries can be a selection from a list of options, e.g.,

```
CHOOSE A BUTTON PRESSING ACTION:
  "KEYBOARD", "FUNC_KEY", "MENU", "VALUATOR"? menu
```

or attributes which are to be included or not included in the interface, as in

```
CAN THE USER UNDO THE PREVIOUS TRANSFORMATION?
  "YES" OR  "NO"? no
```

Note that these queries correspond directly to productions in the universal grammar in Figure 2.35. The interrogate mode creates a subset of the universal action language to describe a personalized user interface. The interpreter for this language is also a subset of the semantic routines already developed as part of GIGL. In the interact mode, a user interacts with the system with a particular personalized user interface specified in the interrogate mode.

```
<file response> ::= "system locates file" | "system indicates
                      no file located" <RECALL>

<ROTATE> ::= <picking action> ["system prompts for angle of
             rotation"] <valuating action> ["system prompts
             for point of rotation"] <locating action>

<MOVE> ::= <picking action> <locating action>

<SCALE> ::= <picking action> ["system prompts for scale factor"]
            <valuating action>

<PARAM> ::= ["system prompts for line style"] <b.p.a.>
            ["system prompts for intensity"] <b.p.a.>

<DEFLT> ::= <system placebo>

<UP> ::= "system identifies higher subpicture in hierarchy" |
         "system indicates no higher subpicture"

<DOWN> ::= "system identifies lower subpicture in hierarchy" |
           "system indicates no lower subpicture"

<EXECUTE> ::= "system deactivates execute/ cancel primitives"
              "system activates all other commands"

<CANCEL> ::= "system deactivates execute/ cancel primitives"
             "system activates all other commands"

<BACKUP> ::= "system identifies previous state of system" |
             "system indicates backup limit reached"
```

Figure 2.35: Portion of a Universal Interaction Language

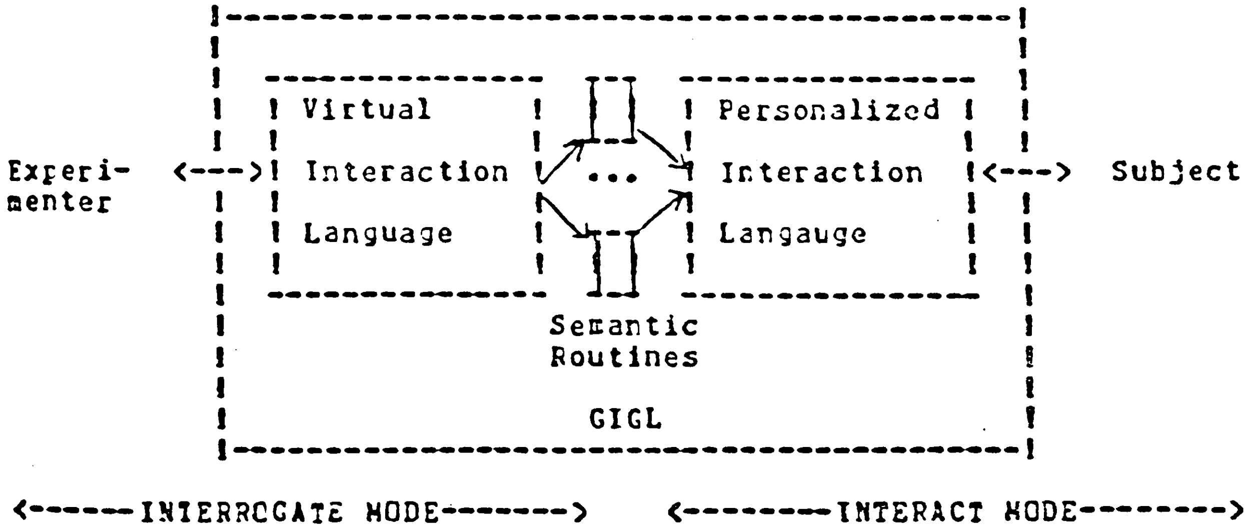


Figure 2.36: Functional Structure of GIGL

CHAPTER 3

MODELING THE MIMD USER INTERFACE DEVELOPMENT ENVIRONMENT

The effectiveness of an interactive software system is dependent on its usability. A software system has maximum usability if it maximizes the productivity of all its end users. The major goal of interactive software engineering is to develop effective software systems to maximize both their usability as well as end user productivity. There are two features of an interactive software system that govern its effectiveness and the productivity of its end users. The first concerns the ability of the system to increase the productivity of end users according to their experience. The second concerns the speed and ease with which end users can utilize new technology that is introduced into the system. New technology is concerned with enhancing any part of the interactive system such as the efficiency of the algorithms and code, the functionality of the system, and the facilities for interaction. Note that both features are dependent on the ability to modify the user interfaces of interactive systems.

The interactive systems that have been developed to date provide limited or no support for modification of user interfaces. Thus, end users are usually forced to learn and master the user interface of every interactive system that they use. Since these interactive systems provide only limited "hard wired" choices for user interface modification, end user productivity can never be maximized. Furthermore, the initial learning curve is high and costly and is the main factor that hinders users from embracing newer tools as well as newer versions of familiar tools.

The basic premise of the MIMD framework is that maximizing user productivity and the effectiveness of interactive systems is entirely dependent on the ability of the end user to define and modify user interfaces. Thus, the major goal of UIDEs within the MIMD framework is to provide facilities that allow end users to develop personalized user interfaces for any interactive application, and to modify any aspect of these personalized interfaces at any time of their experience with the interactive system. Another perspective of the goal of MIMD UIDEs is that the facilities provided to end users are similar to those provided to user interface designers by SISD and SIMD UIDEs. However, MIMD UIDEs differ from UIDEs within other frameworks in the following major ways.

[1] The facilities provided for user interface specification and modification must be usable by end users, who have no knowledge about the implementation of the particular application or the user interface.

[2] The MIMD UIDE is independent of the environment that is used to develop the application component of interactive systems.

Thus, the major problem associated with the development of MIMD UIDEs is that it necessitates a complete and clean separation of the application component from the user interface component of interactive software systems. Such a complete and clean separation facilitates the development of multiple personalized user interfaces for any application. MIMD UIDEs necessitate information describing the characteristics of the application for which the user interfaces are to be constructed or modified. The MIMD UIDE specification mechanism uses this information to construct and modify user interfaces for the particular application.

A clean and complete separation of the application from the user interface necessitates the development of appropriate models for both components. The following subsections develop the MIMD model for applications and user interfaces respectively.

3.1. A Model for Interactive Applications

The main emphasis of the application model is to generalize interactive applications to allow the MIMD UIDE to handle multiple applications in a consistent manner. The application model must also be concerned with modeling the interface between the application and user interface components. This interface necessitates appropriate models for the flow of information from the application to the user interface, and for function invocation and parameter specification.

The **editor** model is a popular and appropriate generalization for interactive applications. The editor model is defined as follows.

[a] A set of **objects** defines the base entities of the application. These objects are combined to form a **picture.** The editor allows the manipulation of this picture in terms of these base objects. There are two types of pictures: **unstructured** and **structured.** An unstructured picture is made up of an arbitrary combination of the base objects. In a structured picture, only certain combinations of base objects are allowed. Note that the possible combinations of base objects can be defined appropriately by a language. Thus, in the case of a structured picture the **terminals** and **nonterminals** of the language define the base objects of the editor, while the **productions** of the language define the manner in which these base objects can be combined to form the picture.

[b] The editor maintains the picture internally within an appropriate data structure. This **major data structure** defines the combination of the base objects that describe the current picture. For structured pictures, this major data structure is usually a syntax tree that must at all times conform to the language definition.

[c] The editor provides a set of **transformational functions** for manipulating the picture. The base transformational functions are **insert** and **delete.** These base functions allow for inserting an (a set of) object(s) into the picture and deleting an (a set of) object(s) from the picture, respectively. Note that all other manipulations (such as replication, substitutions, global deletion/insertion, etc.) can be defined in terms of these base functions. Thus, the base transformational functions must be defined for every editor. In the case of structured pictures, any manipulation of the picture must conform to the language definition. Therefore, the transformational functions must allow only operations that maintain the integrity of

the picture.

[d] The editor also provides a set of **operational functions** that do not change the picture. These operational functions provide the facilities to allow the editor to operate smoothly. Typical operational functions are:

[1] **help:** to provide information about the application's functionality,

[2] **save:** to save the active picture on backing store, and

[3] **load:** to load a previously saved picture from backing store into the editing environment.

It is important to note that a large percentage of currently available interactive systems can be defined using the above editor model.

The next two subsections are devoted to developing models of the two entities of the editor involved in the interface between the application and user interfaces: data structures and functions.

3.1.1. Modeling Interactive Data Structures

The main source of communication between the application and user interface components is the information base of the application component. This information base is managed and maintained within the major data structure of the application component. The internal representation of these data structures must be translated into external forms suitable for viewing by the user. For example, language-based editors maintain a program being edited as a syntax tree. This syntax tree must be translated into an appropriate textual or graphical representation for presentation to the user. Furthermore, the user manipulates the external form, thereby necessitating a translation from the external representation to the internal representation as well. Thus, in the realm of interactive software systems the external representation of data structures is as basic and crucial to the design as its internal representation.

In the conventional model for interactive systems, data structures form the major bottleneck in achieving a complete separation of the application from the user interface of interactive systems. Application information is managed and maintained within conventional data structures that do not provide any facilities to manage the external representations. However, interaction is done solely on the basis of external representations. This mismatch of representations has, to date, tightly coupled the two components leading to the following unresolved issues in user interface research [71]:

[a] how should the user interface access the application's data structure,

[b] should the user interface maintain its own copy of the application's data structure to handle error recovery and undo operations,

[c] how are user picks translated to entities within the application's data structure, and

[d] how is the application's data structure translated to appropriate external representations for presentation to the user.

This dissertation takes the position that the above problems arise due to the inadequacy of conventional data structures in handling the added complexity of interaction.

The design of data structures for interactive systems is crucial as it forms the basis of communication between the two components and thus plays an important role in the separation of the application and the user interface. **Interactive data structures** must be forced to play dual roles, their conventional role in the management and maintenance of data internal to the application, and their role in the management and maintenance of forms of the internal data suitable for processing by entities external to the application component. This duality of data structures, their **internal** and **external** representations, is not a new concept; most software systems provide facilities to translate the internal representation of active data structures into a representation that is suitable for storing on secondary storage. However, managing and maintaining both representations as one abstract object is both an exciting and powerful concept. Note that to be effective, it is desirable that these dual representations be mirror images; that is, there exists an analogy between them such that manipulations of one representation directly translates to manipulations of the other.

The power of such a concept is most evident when we consider the interactive data structure's role in the context of interactive software. Since both the internal and external representations are maintained and managed as a single unit, a complete separation of the user interface component from the application component is possible. The data structure is managed and maintained by the application component. The external representation of the data structure is the basis of communication between the two components. The user interface displays the external representation to the user. User interaction, particularly parameter specification, is in terms of this external representation. The application, through the data structure, can easily translate between entities in the external representation and entities in the internal representation.

The following subsections develop an appropriate model for interactive data structures. The first subsection presents a model for the internal representation of abstract data structures. The next subsection discusses the special requirements of data structures for interactive software, and develops a model for the external representation. The internal and external representations are combined to formulate a comprehensive abstract data structure model for interactive software in the following subsection. The last subsection presents an example of interactive data structures supporting multiple external views.

3.1.1.1. The Internal Representation

This section develops a model to describe the internal representation of abstract data structures. By internal representation we imply the form of data structures that are active within main memory during the execution of the software system. The framework that is employed to describe an abstraction describes the information local to the abstraction and the set of abstract operations that is provided.

As the term implies, data structures are composed of two objects, the **data** and the **structure** that is imposed on the data. The **cell** or **node** is the basic building block of data structures. Thus data structures are a collection of nodes connected according to the structure definition. In its most general form, a node contains one or more fields. Each field is defined appropriately by a name that identifies the field and its corresponding type. Thus each node has the form

{<field_name> : <field_type>}+

where the '+' indicates one or more occurrences.

The definition of a node describes the local information of the data abstraction. The set of operations defined for the data abstraction describe the behavior of nodes. We can identify four basic operation types that must be defined for any node:

[1] the **assignment** operation which allows values to be assigned to nodes,

[2] the **retrieval** operation which allows values of fields to be retrieved,

[3] the **comparison** operation to allow nodes to be compared, and

[4] the **attribute** operation which allows queries on the attributes of the node (e.g., size, types of fields, etc.).

Of course, a node may extend these base operations to define operations that are specific to its nature. For example, type coercion, that is the ability to change a given type to the node's type, is a special operation which can only be defined for certain classes of nodes. Another example are relational comparison operations (<, <=, >, >=) which can only be defined for nodes which have a definite ordering. Other operations are addition, concatenation, subtraction, union, etc. However, the models developed for abstract interactive data structures will only consider those operations that are basic to every instance.

The structure abstraction is concerned with defining the relation of data nodes and the set of operations that define the behavior of the structure, or more specifically the manner in which data nodes are managed and maintained. The local information of the structure abstraction defines **neighborhood relationships** which describe how nodes are connected to neighboring nodes in the structure. The basic set of operations that must be defined for each structure are as follows.

[1] **Access** operations to allow nodes in the structure to be accessed. Access operations can be further refined into the following two types:

[a] a **next** operation which allows the neighbors of a particular node to be accessed, and

[b] a **traversal** operation which uses the next operation to access all the nodes in the structure in a specific order.

[2] **Search** operations that provides facilities to identify a node that meets certain criteria (e.g., a specific value of a field). Note that search operations are dependent on the comparison operation provided by data nodes in the structure.

[3] **Manipulator** operations that allow the structure itself to be manipulated. In particular, manipulator operations allow for insertions and deletions of nodes to and from the structure. Manipulator operations must ensure that the structure is always consistent. That is, the resulting structure does not violate the neighborhood relationships defined between nodes.

Figure 3.1 summarizes the model for the internal representation of data structures. This model is general enough to describe most data structures. If the node information contains a single field of a base type (i.e., integer, real, character, string), then the operations are exactly those that are defined for these base types in any high-level programming language. For nodes that contain composite fields, the operations will depend on the nature of the fields. Most of the operations can be defined in terms of operations defined for the base types. For instance, a particular field may be designated the **key** and the comparison operators defined in terms of the

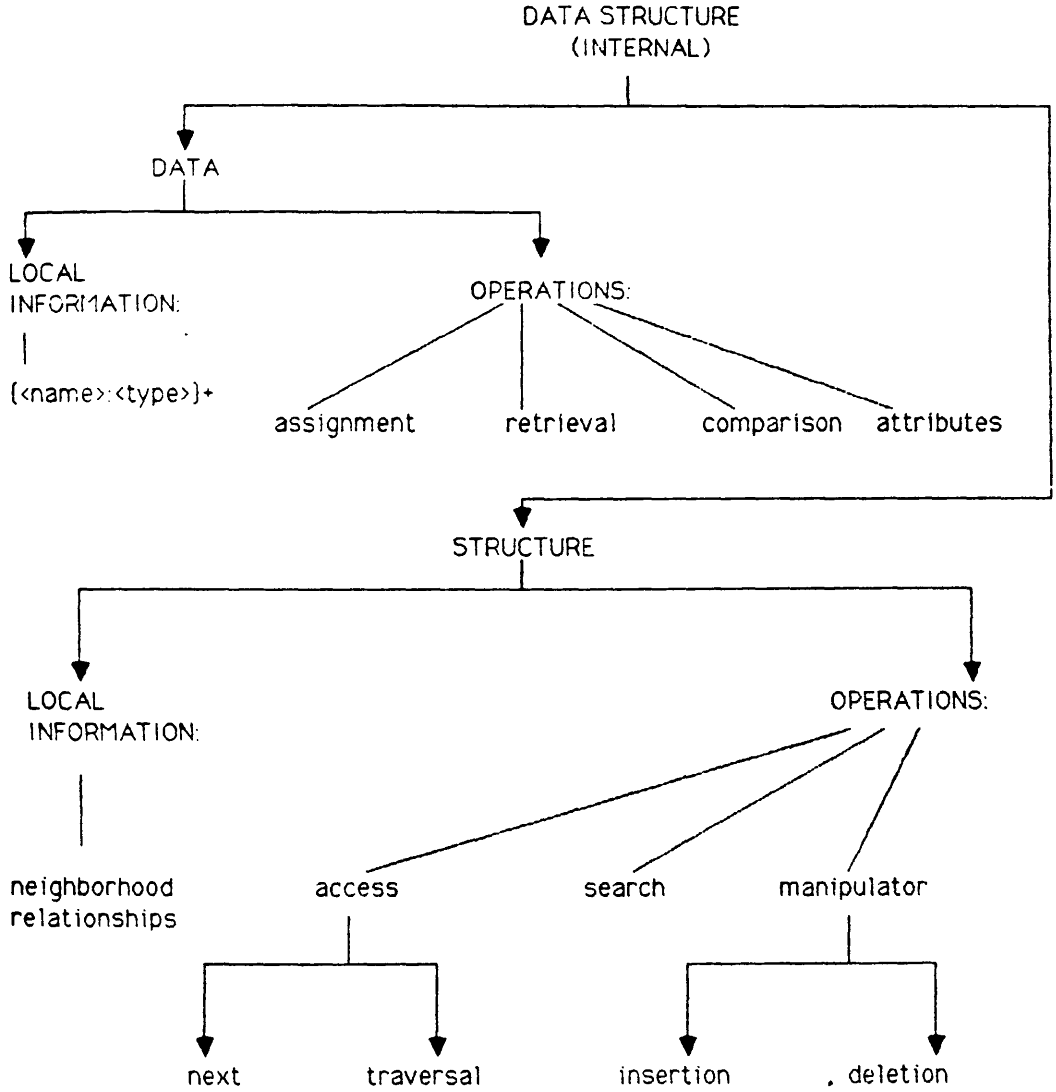


Figure 3.1: A Model for the Internal Representation

comparison operators defined for the type of the key field. As examples of the structure abstraction, consider the definition of lists and binary trees as depicted in Figures 3.2 and 3.3 respectively.

3.1.1.2. The External Representation

Before we present a model for the external representation, it is appropriate to outline our assumptions about the underlying graphics capabilities. In general, an external representation can be modeled as an arbitrary two-dimensional graphical display which is maintained in an internal **display buffer.** Conceptually, display buffers are infinite allowing the entire display of any given data structure, however large, to be generated. A graphics package provides the facilities (e.g., routines for drawing lines, curves, etc. and graphic primitives such as color, reverse-video, etc.) to allow an arbitrary two-dimensional picture to be generated in the display buffer. Thus, the model of each node of the data structure defines an appropriate **display routine** which describes its external form. Each display routine is passed entry coordinates into an appropriate display buffer on invocation and has the effect of generating its external image at the specified position in the display buffer.

Due to the inherent duality of representations, conceptually it should be possible, and it is desirable, to model the external representation of data structures using the same framework that was used to model its internal representation as summarized in Figure 3.1. The model for the external representation is shown in Figure 3.4. Note the similarity between the two figures. The external representation model is also made up of a data abstraction and a structure abstraction.

For each node in the internal representation, there exists a node in the data abstraction of the external representation. This node, which defines the local information of the data abstraction, contains the display routine and its corresponding display attributes (e.g., width and height) of the external image of the node. The four basic operations defined for the internal representation are redefined in the following manner.

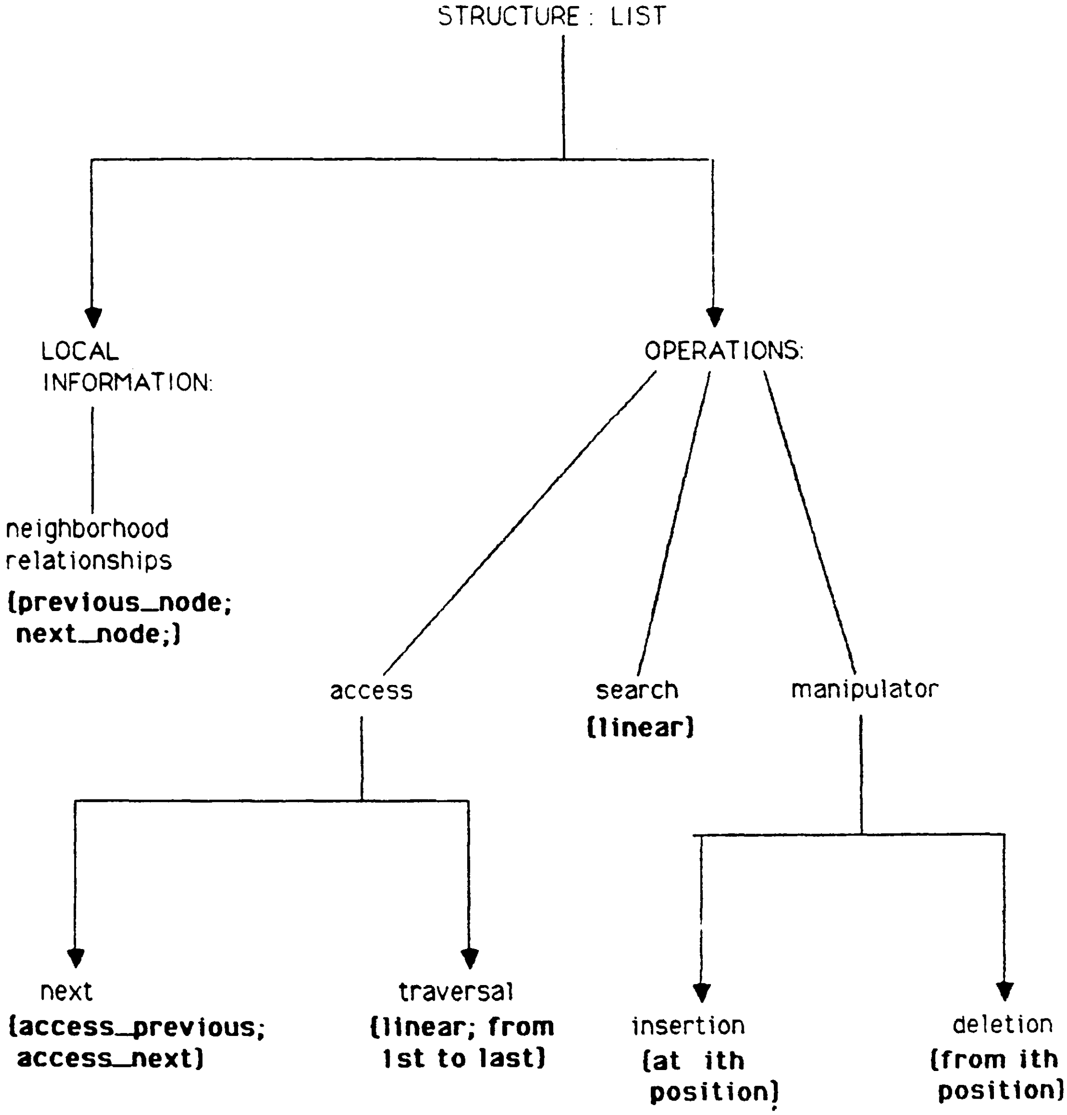


Figure 3.2: The Internal Representation of Lists

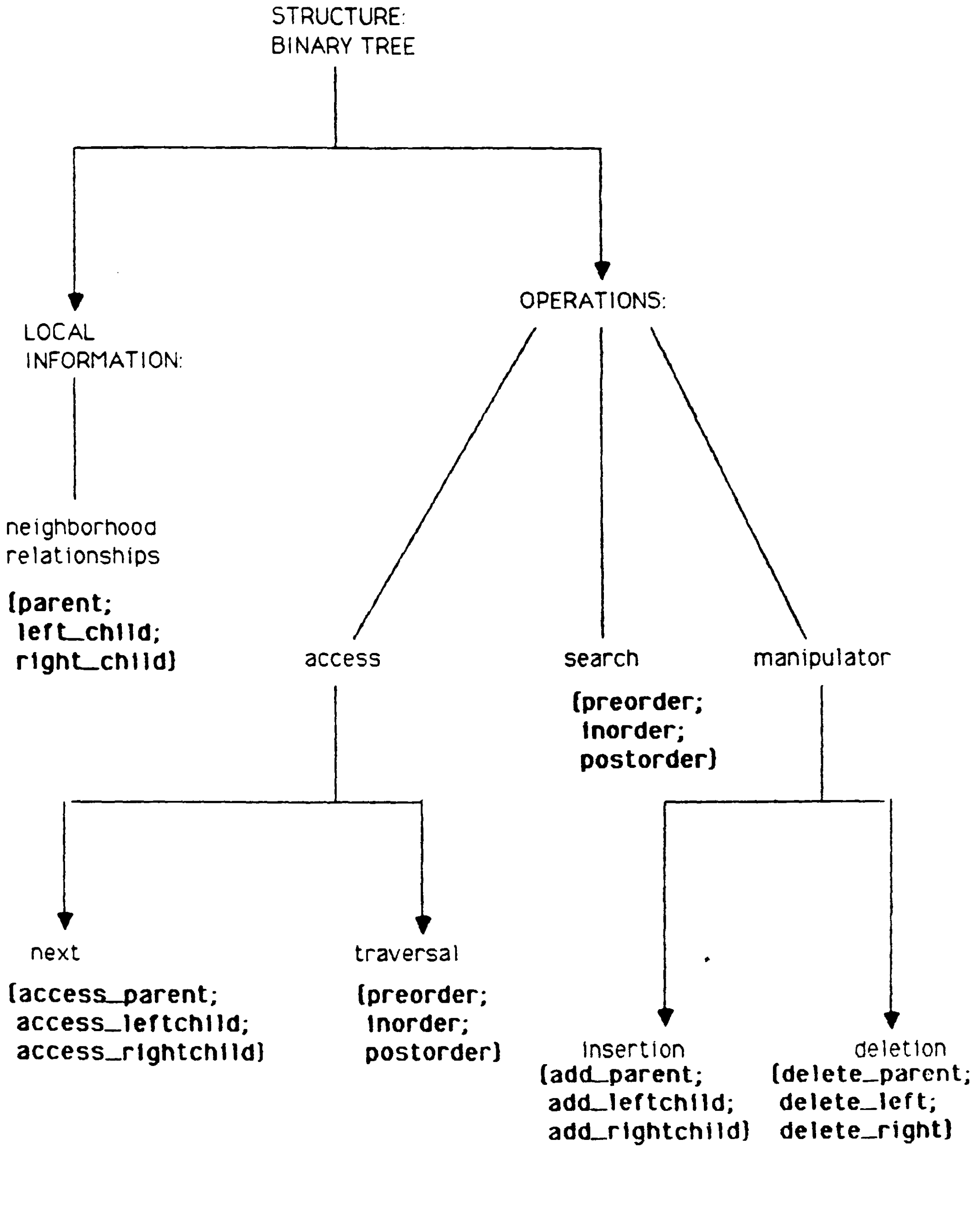


Figure 3.3: The Internal Representation of a Binary Tree

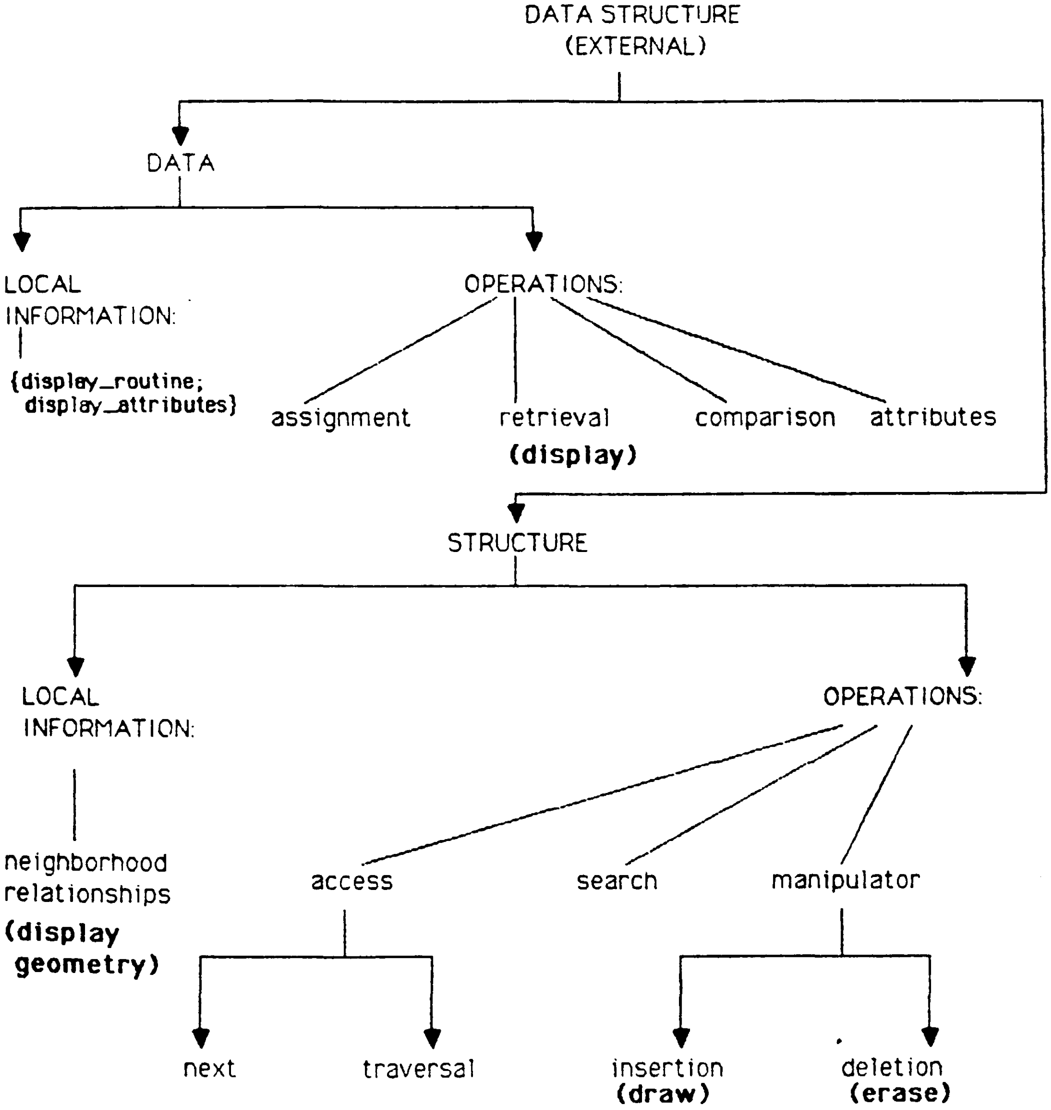


Figure 3.4: A Model for the External Representation

[1] The assignment operation allows the display routine along with its display attributes to be associated with a node.

[2] An important retrieval operation is the invocation of the display routine to facilitate the generation of the node's image.

[3] The comparison operation is used to determine whether a particular (x,y) coordinate lies within the display of the node. This operation facilitates the identification of a node through its external display.

[4] The attribute operation allows for queries of display attributes (e.g. the width and height of the display).

The structure abstraction defines the behavior of the display buffer which will contain the external representation of the data structure. The local information of this abstraction defines the relationship of each node's display to the displays of its neighboring nodes. That is, it describes the geometry of the display of the entire structure. More specifically, the neighborhood relationships define the necessary information to calculate the entry coordinates of each node's display in the display buffer. Note that this calculation necessitates knowledge of the width and height of each node's display which is readily available from the data abstraction. A similar set of operations that were defined for the internal structure abstraction is provided.

[1] The access operation is defined by the next and traversal operations. The next operation allows access of the displays of neighbors of a particular node's display. The necessity of such an operation is best described by the following scenario. The external representation of the data structure is presented to the user on some output device. A cursor is positioned on some part of this external display, that is, within the display of some node in the external representation (which can easily be determined from the comparison operation provided by the data abstraction). The user now indicates that the cursor be moved to the next node's (vertically below the current node) display. This action is translated to a call on the next operation of the structure abstraction which calculates and returns the entry

coordinates of the display of the node directly below the current node. The traversal operation allows an ordered access of each node's display in the buffer.

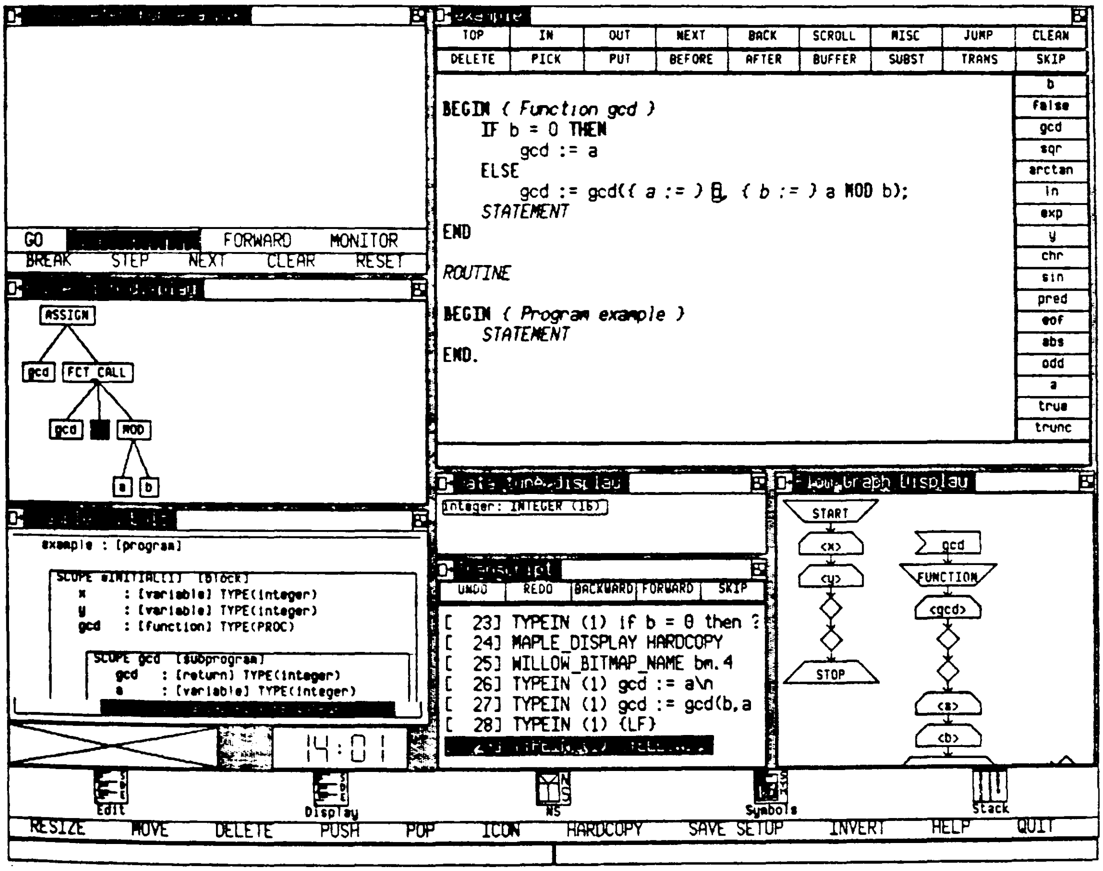
[2] The search operation facilitates the identification of a node according to its external display. Specifically, the search operation accepts an (x,y) coordinate in the display buffer and identifies the node within whose display the (x,y) coordinates lie. The search operation is easily described in terms of an ordered traversal which accesses each node's comparison operation. This is a useful operation to facilitate parameter specification.

[3] The manipulator operations allow for quick insertions (draws) and deletions (erasures) of a node's display to and from the display buffer respectively.

3.1.1.3. A Comprehensive Model

It is important to note the similarity of the models that define the internal and external representations of data structures. This solidifies the concept of the close relationship between the two representations. However, to arrive at an effective comprehensive model for interactive data structures we must merge the internal and external models to allow the two representations to be maintained and managed as a unit.

This comprehensive model of interactive data structures is further complicated by the requirements of the more sophisticated interactive systems which rely on **multiple** external representations of an internal data structure to increase the bandwidth of communication between the user and the internal information maintained by the application. As an example, the popularity and effectiveness of the many language-based structured editors available today is based on the ability of allowing the user to view and manipulate a single program as a collection of external views, for example, a textual view, a programming language view (e.g. Pascal, C, Ada, etc.), a graphical view (e.g., Nassi-Schneiderman diagrams, flow charts, etc.), an execution view, etc. Figure 3.5 presents a typical example ([83]) of such an editor's interface. Note that the user can manipulate any of these views, and these manipulations must be translated automatically by the system to manipulations of the single internal representation of the program, usually



```
BEGIN { Function gcd }
    IF b = 0 THEN
        gcd := a
    ELSE
        gcd := gcd({ a := } b, { b := } a MOD b);
    STATEMENT
END

ROUTINE

BEGIN { Program example }
    STATEMENT
END.
```

```
example : [program]

SCOPE $INITIAL[1]  [block]
    x      : [variable] TYPE(integer)
    y      : [variable] TYPE(integer)
    gcd    : [function] TYPE(PROC)

    SCOPE gcd  [subprogram]
        gcd    : [return] TYPE(integer)
        a      : [variable] TYPE(integer)
```

```
integer: INTEGER (16)
```

```
[  23] TYPEIN (1) if b = 0 then ?
[  24] MAPLE_DISPLAY HARDCOPY
[  25] WILLOW_BITMAP_NAME bm.4
[  26] TYPEIN (1) gcd := a\n
[  27] TYPEIN (1) gcd := gcd(b,a
[  28] TYPEIN (1) {LF}
```

Figure 3.5: An Editor with Multiple External Representations

in the form of a syntax tree. Conversely, since all the external views are generated from the internal syntax tree, modifications of the syntax tree must be automatically and immediately reflected in all its external views.

Therefore, the model for interactive data structures must allow for a single internal representation and its multiple external representations to be managed and maintained as a unit. Figure 3.6 presents such an integrated model for interactive data structures. Note that the structure manipulator operations are redefined so that any changes made to the data structure automatically updates all its external representations.

3.1.1.4. Example of Interactive Data Structures

To close this section, let us consider an example of an interactive system and design its interactive data structure. Suppose we would like to build an interactive system, EMPSYS, to manage the employees of a company. The company is divided up into a number of departments. Each department has a manager who is in charge of a number of employees. A Chief Executive Officer (CEO) sits on top of the employee hierarchy and is in charge of all the managers.

EMPSYS allows the manipulation of employees of a company. The user of EMPSYS is allowed to access employees in the following three ways:

[1] by their social security number,

[2] by their name, and

[3] by their position in the employee hierarchy.

Since the design of the internal representation of the employee data structure is well understood, our objective in the following discussion is to concentrate on the design of the external representations and highlight the correspondence between the dual representations. To simplify the discussion, we use a list as the internal representation of the employee data structure.

INTERACTIVE DATA STRUCTURES

Local Information

(DATA ABSTRACTION)

- [a] NODE INFORMATION:
  - Internal - {<field_name> : <field_type>}+
  - External - {display_func; display_attr}+

(STRUCTURE ABSTRACTION)

- [b] NEIGHBORHOOD RELATIONSHIPS:
  - Internal - linking conventions
  - External - display geometry

Operations

(DATA ABSTRACTION)

- [1] ASSIGNMENT OPERATIONS:
  - Internal - assign value to internal representation
  - External - assign value to external representation
- [2] RETRIEVAL OPERATIONS:
  - Internal - return value of internal representation
  - External - display external representation
- [3] COMPARISON OPERATIONS:
  - Internal - compares two internal data nodes
  - External - compares an (x,y) coordinate with node's display
- [4] ATTRIBUTE OPERATIONS:
  - Internal - returns data attribute
  - External - returns display attribute

(STRUCTURE ABSTRACTION)

- [5] ACCESS OPERATIONS:
  - [a] NEXT OPERATION:
    - Internal - return neighbor of data node
    - External - return neighbor of node's display
  - [b] TRAVERSAL OPERATION:
    - Internal - access each node's data
    - External - access each node's display
- [6] SEARCH OPERATIONS:
  - Internal - search for node with matching data
  - External - search for node whose display contains (x,y) coordinates
- [7] MANIPULATOR OPERATIONS:
  - [a] ADDITION OPERATION:
    - Internal + External - adds new node in data structure and updates all external representations
  - [b] DELETION OPERATION:
    - Internal + External - deletes node from data structure and updates all external representations

Figure 3.6: An Abstract Model For Interactive Data Structures

The internal representation of the data abstraction defines the following fields for each node in the data structure:

[a] the name of the employee (NAME),

[b] the employee's social security number (SS#),

[c] the position of the employee in the company (POS) which can be one of {CEO, Manager, Employee},

[d] the name of the department where the employee works (DEPT), and

[e] fields that describe the control hierarchy of employees within the company.

This control hierarchy is described by the following fields for the employee (CEO or Manager) that is in control:

[a] the number of employees that the controller is in charge of (CONTROLS),

[b] the identifier of the first employee that is controlled (FIRSTEMP), and

[c] the identifier of the last employee that is controlled (LASTEMP).

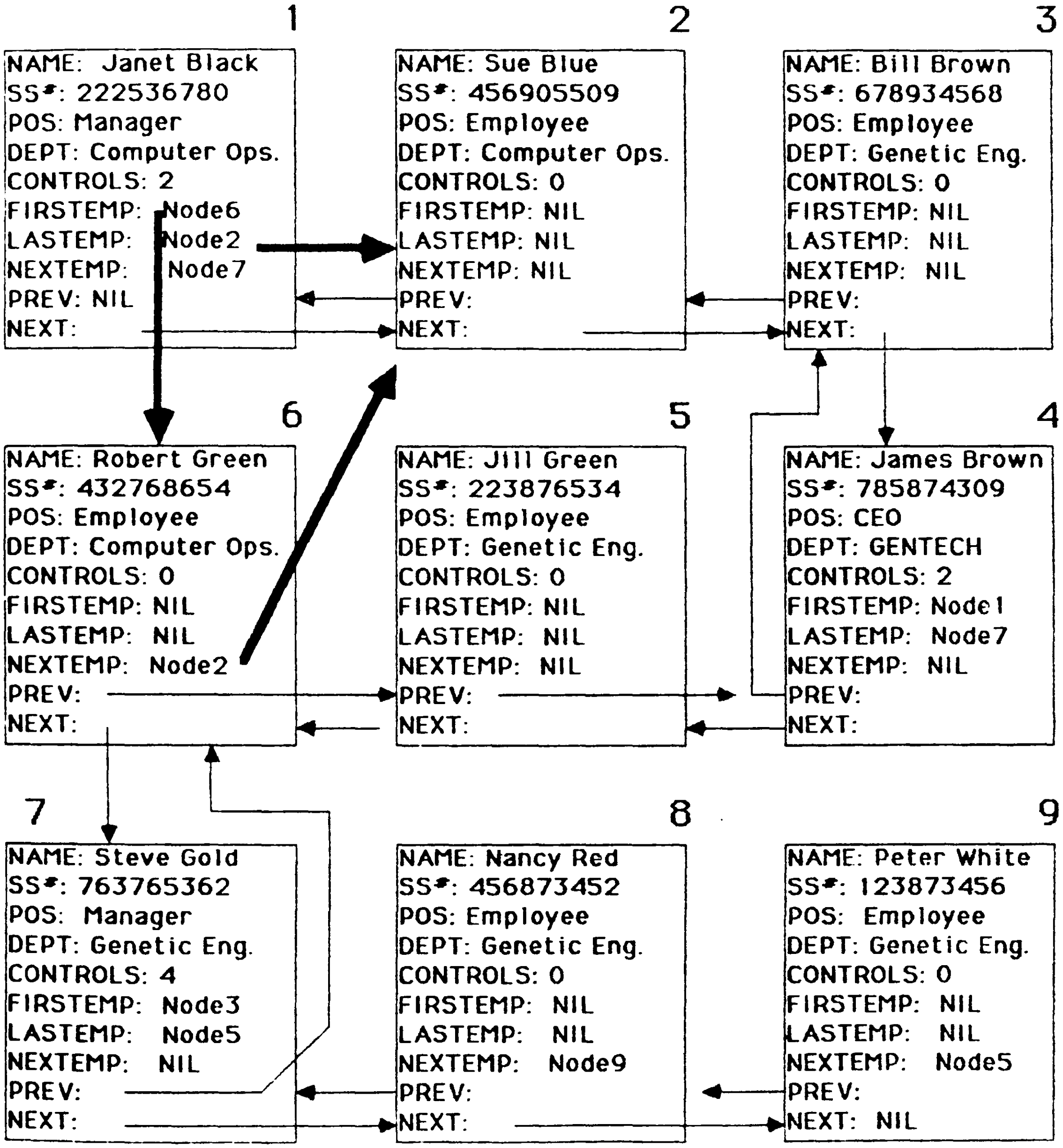
To complete the description, the following fields are defined for employees that are controlled:

[a] the identifier of the next employee in the control chain (NEXTEMP), and

[b] the identifier of the controlling employee (BOSS).

The structure abstraction, the list, contributes the PREV and NEXT fields that defines the neighborhood relationships of nodes in the list. Figure 3.7 presents an example of the internal representation. The thick arrows in the Figure depict the control chain for the employees within the **Computer Operations** department of the company.

According to the specification, the interactive data structure must maintain three external representations: **name_view, ss#_view,** and **position_hierarchy_view.** The external representation therefore contributes a set of three display functions (along with their associated attributes) to the definition of each node in the interactive data structure. These display functions are



1

NAME: Janet Black
SS*: 222536780
POS: Manager
DEPT: Computer Ops.
CONTROLS: 2
FIRSTEMP: Node6
LASTEMP: Node2
NEXTEMP: Node7
PREV: NIL
NEXT:

2

NAME: Sue Blue
SS*: 456905509
POS: Employee
DEPT: Computer Ops.
CONTROLS: 0
FIRSTEMP: NIL
LASTEMP: NIL
NEXTEMP: NIL
PREV:
NEXT:

3

NAME: Bill Brown
SS*: 678934568
POS: Employee
DEPT: Genetic Eng.
CONTROLS: 0
FIRSTEMP: NIL
LASTEMP: NIL
NEXTEMP: NIL
PREV:
NEXT:

6

NAME: Robert Green
SS*: 432768654
POS: Employee
DEPT: Computer Ops.
CONTROLS: 0
FIRSTEMP: NIL
LASTEMP: NIL
NEXTEMP: Node2
PREV:
NEXT:

5

NAME: Jill Green
SS*: 223876534
POS: Employee
DEPT: Genetic Eng.
CONTROLS: 0
FIRSTEMP: NIL
LASTEMP: NIL
NEXTEMP: NIL
PREV:
NEXT:

4

NAME: James Brown
SS*: 785874309
POS: CEO
DEPT: GENTECH
CONTROLS: 2
FIRSTEMP: Node1
LASTEMP: Node7
NEXTEMP: NIL
PREV:
NEXT:

7

NAME: Steve Gold
SS*: 763765362
POS: Manager
DEPT: Genetic Eng.
CONTROLS: 4
FIRSTEMP: Node3
LASTEMP: Node5
NEXTEMP: NIL
PREV:
NEXT:

8

NAME: Nancy Red
SS*: 456873452
POS: Employee
DEPT: Genetic Eng.
CONTROLS: 0
FIRSTEMP: NIL
LASTEMP: NIL
NEXTEMP: Node9
PREV:
NEXT:

9

NAME: Peter White
SS*: 123873456
POS: Employee
DEPT: Genetic Eng.
CONTROLS: 0
FIRSTEMP: NIL
LASTEMP: NIL
NEXTEMP: Node5
PREV:
NEXT: NIL

Figure 3.7: The Internal Representation for EMPSYS

capable of generating the appropriate display of the node in each of the three external representations that are maintained. Note that the interactive data structure must maintain three display buffers to hold the three external views. Figure 3.8 extends the example in Figure 3.7 by combining the internal representation with the three external representations resulting in a single interactive data structure. In the Figure, display functions are abbreviated to **d_fi** and display attributes to **d_ai** where "i" stands for the ith external representation. The Figure highlights the boundaries of nodes' displays within each buffer.

The neighborhood relationships of the structure abstraction of each external representation provides the information for calculating the entry coordinates of each node's display within the appropriate display buffer. For example, the following information calculates the entry coordinates for each node's display within the name_view display buffer

(x,y) = previous node's entry coordinates;

entry coordinates = (x, y + previous node's height + K);

where K is a constant to allow for some space between displays. The first node's display is appropriately defined as

entry coordinates = (0, display buffer's width / 2).

As an example of interacting with the above data structure, consider the following scenario. The position_hierarchy display buffer has been passed by the application to the user interface to be presented to the user. The user interface presents a portion of the display buffer within a window on some output device. Say the user would like to delete a particular employee from the company. The user specifies the employee to be deleted by moving a cursor with a mouse to the desired node in the screen. These screen coordinates must now be translated by the interactive system to a particular node in the interactive data structure. The user interface has the necessary information to translate the screen coordinates to coordinates within the position_hierarchy display buffer. The search operation provided by the external representation is capable of determining the node within whose display these coordinates lie. Note that the

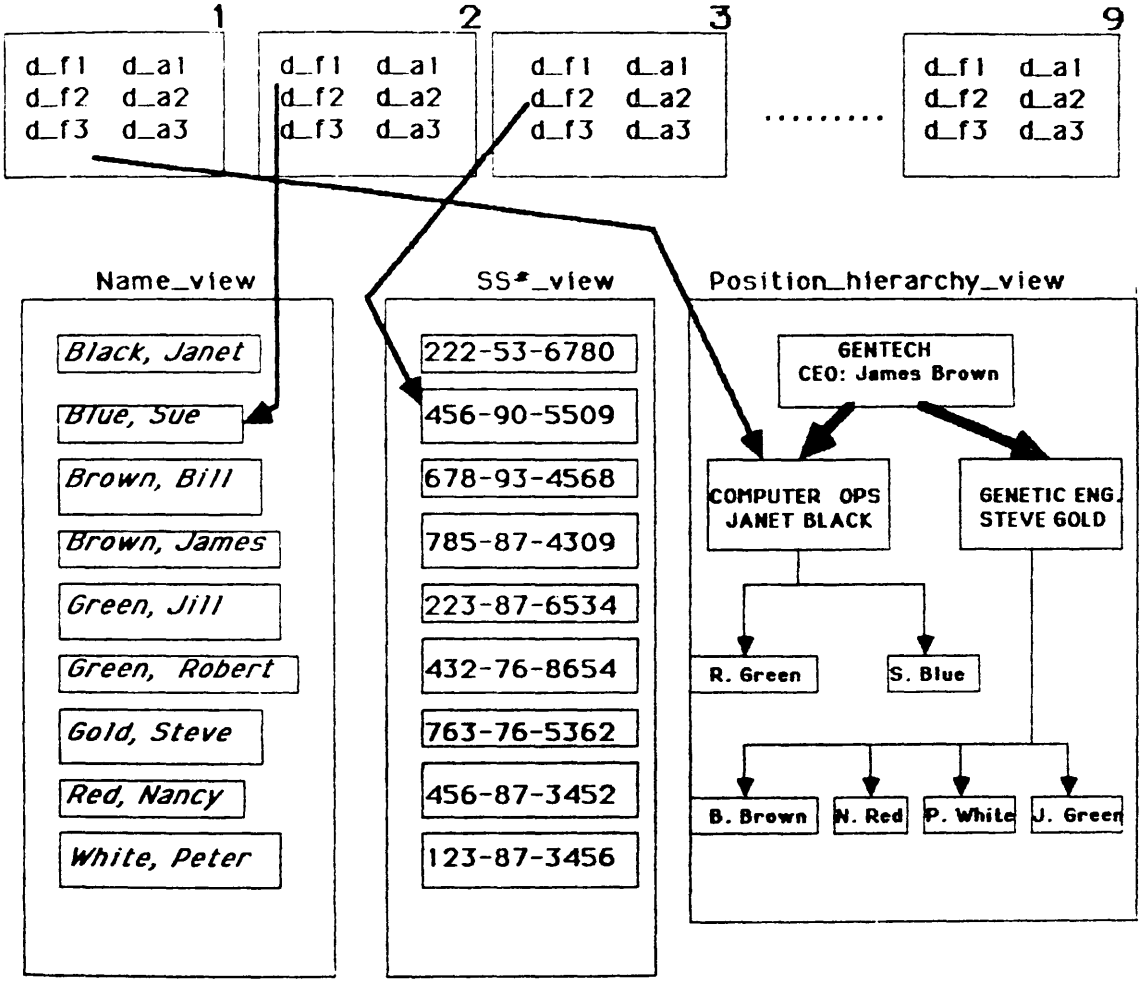


Figure 3.8: The External Representations for EMPSYS

search operation uses the traversal operation to access each node's comparison operation. Finally the node is deleted from the interactive data structure through its manipulator operation which has the effect of updating all of the (three) external representations to reflect the change of the deletion.

3.1.2. Modeling Application Functions

The final issue that must be considered to achieve the total separation of the application from the user interface concerns function invocation and parameter specification. In general, each application function requires the specification of a set of parameters and the end result is either error feedback or some information depicting the effect of the function call. However, within an interactive environment, the function is capable of providing error feedback or some resulting information after the specification of each parameter. The error usually indicates an invalid parameter specification. The resulting information could either be help information to aid in specifying the next parameter or information that shows the overall effect of the function call. For example, consider the insert function that requires two parameters, the identification of the point of insertion and the specification of the object to be inserted. The specification of the first parameter could result either in an error indicating an invalid insertion point, or a list of valid objects that could be inserted at the specified insertion point to aid in selecting the second parameter. The second parameter could also result in an error indicating an invalid object for insertion, or the insertion of the specified object at the specified point. The resulting information is the update of the external representations showing the result of the insertion. Thus, each function necessitates interaction with the user. The conventional approach to interactive system design is to allow each function to be involved in parameter specification, that is, accepting user input and displaying intermediate results. The role of the user interface is therefore subordinate to function invocation. Thereafter, the application takes over control of the interaction.

To achieve a complete separation of the application from the user interface, interaction with the user must be the sole responsibility of the user interface. This necessitates a redesign of the functions provided by the application. An appropriate model of interactive functions that allows such a complete separation is as follows.

[a] Each function is defined by a set of **services,** where each service is capable of processing its parameters independently. Usually, the function defines a service to handle each of the parameters that must be specified. On invocation, the function presents the set of services it can perform. The function can also define a default ordering of its services.

[b] On invocation, each service accepts its parameter(s), an **error** buffer and a **result** buffer. Upon completion, the service presents information either in the error buffer indicating an error in parameter specification, or in the result buffer which describes the effects of the service.

Note that such a model allows the specification of the action to be taken upon service completion. This action is either the invocation of a particular service provided by the function or quitting the function. For example, the next action to be taken when an error is detected is usually to call the service again. This then allows the user interface to totally control the application's functions, and more importantly does not require the functions to be involved in user interaction.

3.2. A Model for Customizable User Interfaces

Given the model of interactive applications as presented in the previous subsection, the user interface model is mainly concerned with allowing customization of every object involved in interaction. Interaction can be divided into the following two categories.

[1] **Presenting information to the user:** This category encompasses the interfacing of information to the user. It is therefore concerned with presenting application specific information, as well as user interface information involved with interaction. The major types of information to be interfaced to the user include the external representations of the information base of the application, the base entities of the application, menus, help and error messages, and interaction feedback.

[2] **Function invocation and parameter specification:** This category encompasses the main goal of interaction, that is, the invocation of functions and the specification of parameters.

The following subsections develop models to allow customization of the above two categories of interaction respectively. The final subsection presents the overall user interface model.

3.2.1. Presenting Information to the User

Customization of the information presented to the user is provided by the following two objects of the user interface component.

[1] **Name-mapping objects** provide facilities to define external user-defined representations for any internal entity involved in interaction. Thus, users can define personalized names for any name used in interaction, including function and parameter names.

[2] **Display objects** provide facilities to customize the layout of the information presented to the user.

Note that name-mapping objects are concerned with defining a one-to-one mapping between internal and external representations. They are therefore modeled in exactly the same manner as the internal and external representations of the data component of interactive data structures as presented in section 3.1.1. The following discussion is devoted to modeling display objects.

Display objects control the presentation of information to the user. There are three types of information that the interactive system presents to the user.

[1] The external representations of the major data structure maintained by the application.

[2] Information maintained by the user interface to aid in function invocation and parameter specification. This information (usually) corresponds to menus of functions, services and application entities.

[3] System messages that correspond to error messages or textual feedback about the current interaction.

Thus, display objects are concerned with presenting the above information on some physical output device for viewing by the user. The specification mechanism provides the facilities to allow the user to customize these display objects thereby personalizing the presentation of information.

Any information that is to be presented to the user is maintained internally (either in the application or user interface components) within interactive data structures. Each interactive data structure defines the external representations of its internal form that is suitable for viewing by the user. It is these external representations, which are maintained within internal display buffers, that are interfaced to the user through display objects. In other words, display objects provide the facilities to map information within these display buffers to physical output devices for viewing by the user.

The internal display buffers contain the image of the entire interactive data structure. Display objects provide the mapping from display buffer to output device through **windows** and **viewports.** Windows define rectangular regions within display buffers describing the extent of the image that is presented to the user. Viewports map windows onto rectangular regions defined on physical output devices. [Note that these standard definitions for windows and viewports [84] conflict with the use of the term windows as defined by some current interactive systems (e.g., window managers) to designate an area on the screen (i.e., a viewport)]. Note that by adjusting the position or size of the window relative to the display buffer, the effects of **panning** and **zooming** can be produced. The interactive system can present the largest possible picture of the image within the display buffer by defining a window that just surrounds the image within the display buffer. A smaller picture of the entire image is produced by defining a window larger than the image. Conversely specifying a smaller window forces **clipping** of the image allowing a larger scale of the portion of the image presented to the user. These effects are depicted pictorially in Figure 3.9 (a), (b) and (c) respectively. Note that the size of a viewport controls what portion of the window is depicted on the output device. The entire model of

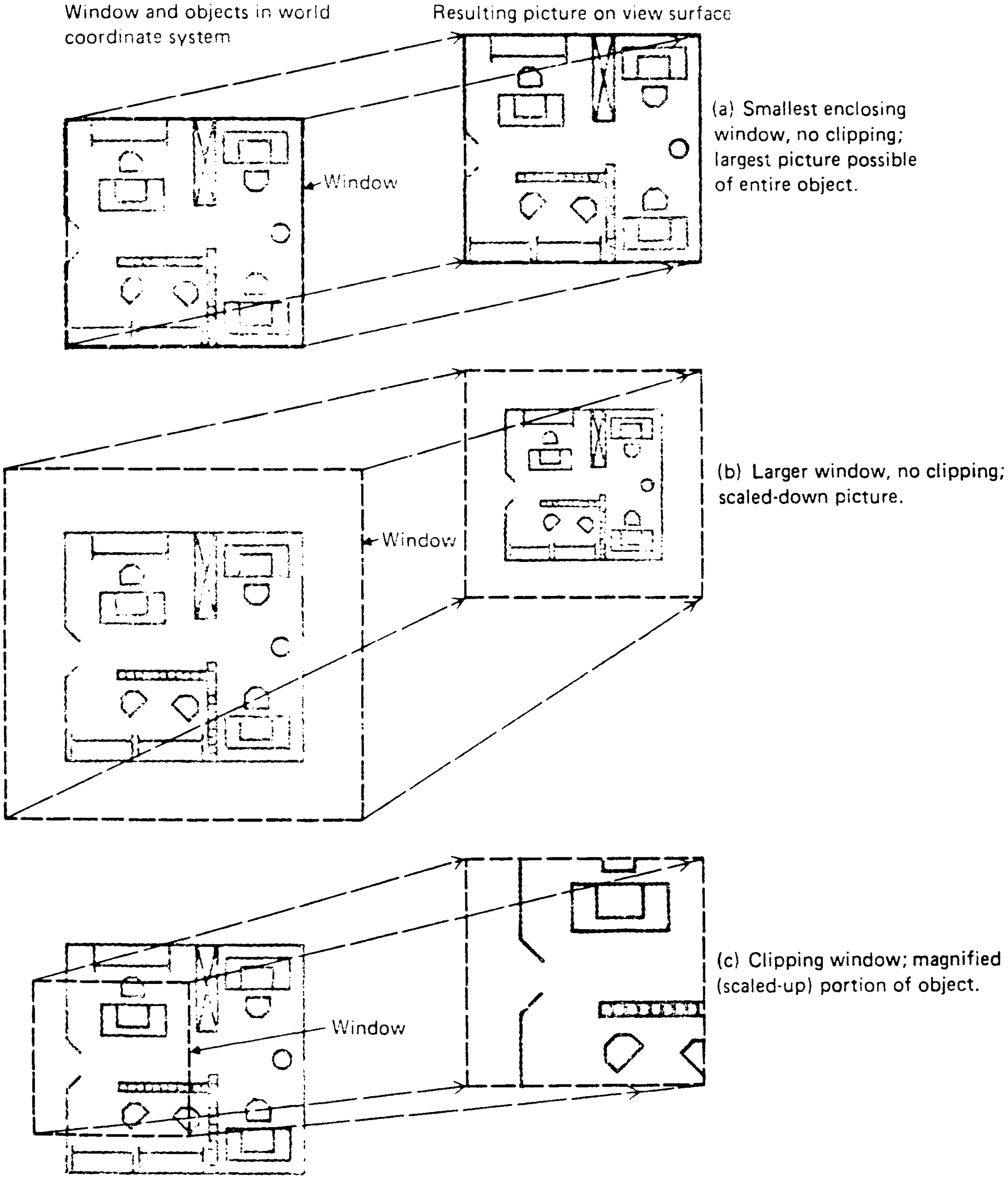

Window and objects in world coordinate system
Resulting picture on view surface
Window
(a) Smallest enclosing window, no clipping; largest picture possible of entire object.
Window
(b) Larger window, no clipping; scaled-down picture.
Window
(c) Clipping window; magnified (scaled-up) portion of object.

Figure 3.9: Mapping Window Contents to Viewing Surface

presenting information to the user is depicted in Figure 3.10.

3.2.2. Function Invocation and Parameter Specification

The most important and most crucial aspect of customizing user interfaces is the description of how the user interacts with the functions provided by the application. Typically, user interaction is described as follows. The user initially activates a function within the application. Once the function is activated, the services provided by the function are called in some specified order. Each service necessitates the specification of some parameter(s), and produces either some error feedback or some information resulting from the service. This information is presented to the user. In either case, the next action to be performed is either a service provided by the function or quitting the function altogether.

The user interface model for functions resembles the application model for functions as presented in section 3.1.2. For each application function a corresponding **function interaction object** is defined within the user interface model. This function interaction object is concerned with function invocation and managing the services provided by the function. Each service of an application function is modeled by a corresponding **service interface object** within the user interface. The service interface object is in charge of interacting with the user to specify the required (set of) parameter(s), calling the associated service within the application, and displaying the results of the service to the user. Figure 3.11 depicts the relationships between function and service objects in the application model and their counterparts in the user interface model.

The final aspect of the user interface model concerns parameter specification which is the major component of interaction. Parameter specification is an **interaction task** which is appropriately specified in terms of **input techniques** and the corresponding **interaction devices.** Foley, Chan and Wallace [14] have identified six fundamental interaction tasks that are application and device independent. These tasks can be mapped onto many different techniques and devices. The fundamental tasks are:

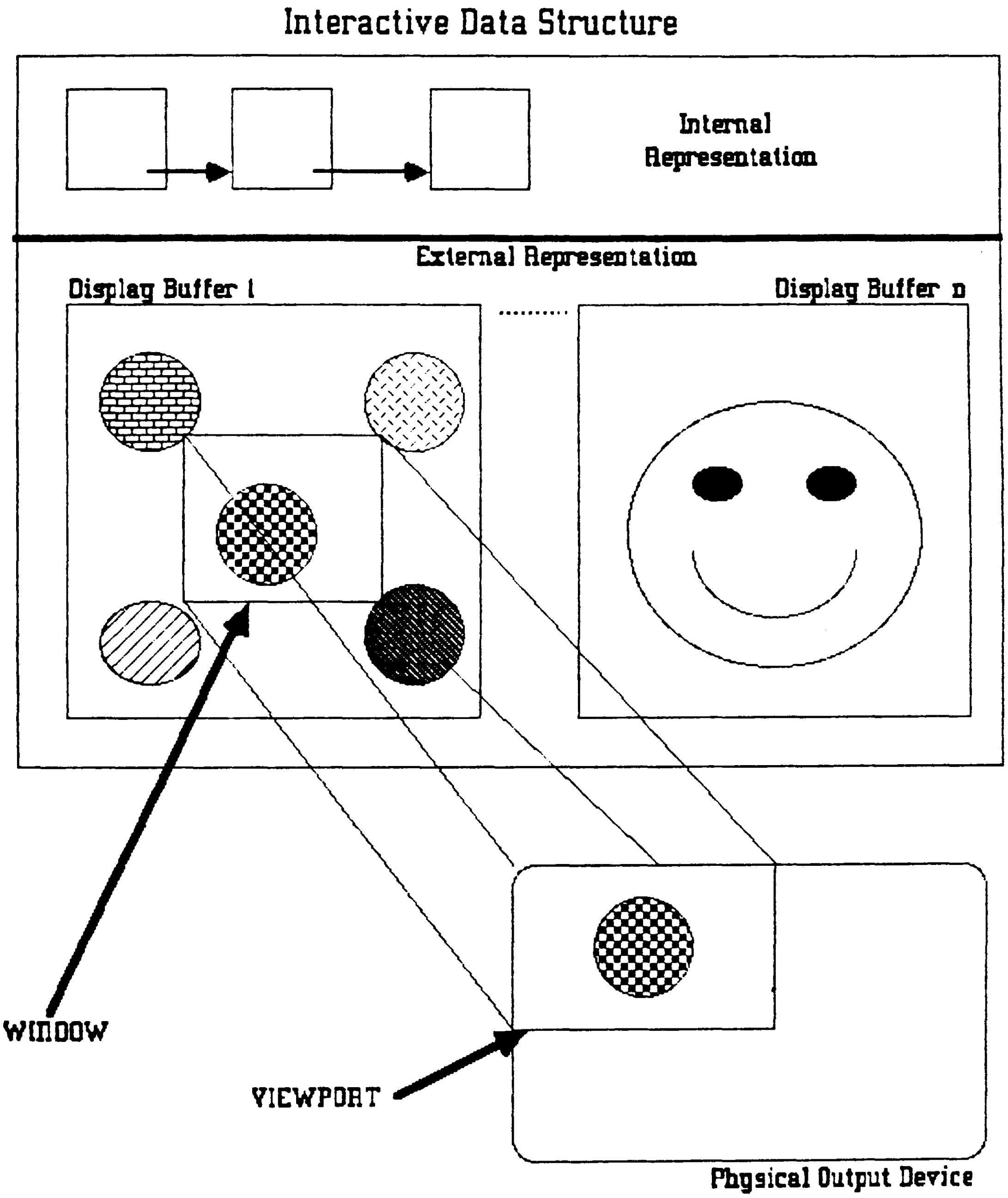


Figure 3.10: Interfacing Information to the User

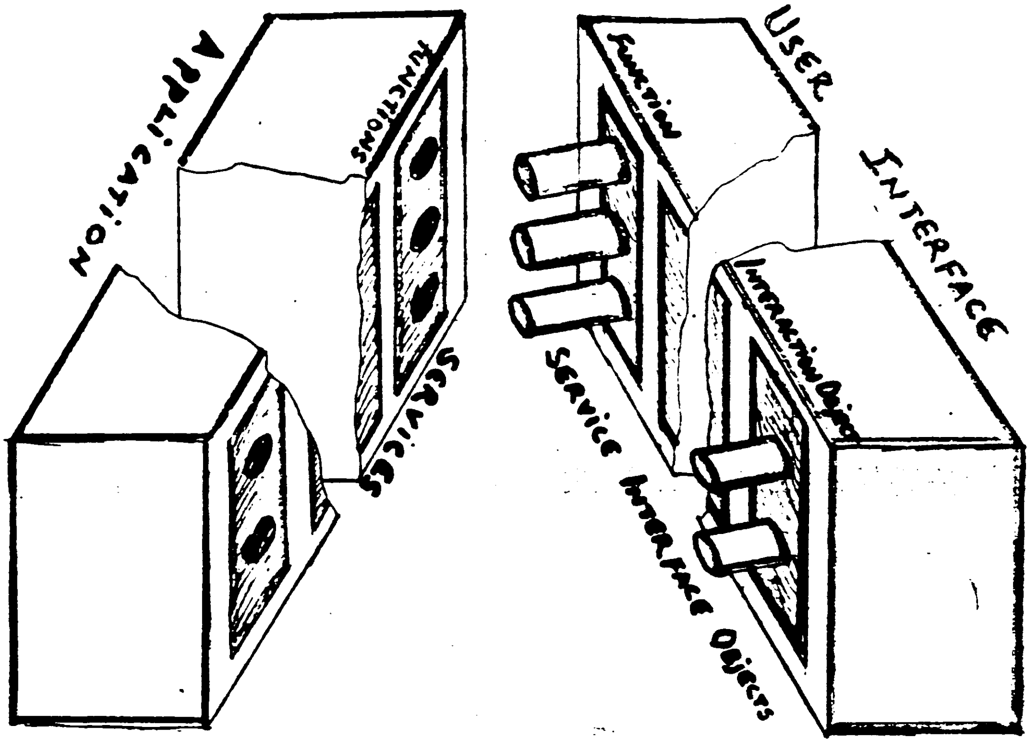


Figure 3.11: Function Interaction Objects

SELECT: select from a set of alternatives (usually) in a menu;

POSITION: indicate a position ((x,y) coordinate) on a output device;

ORIENT: orient an entity in 2-d or 3-d space;

PATH: generate a time sequence of positions or orientations;

QUANTIFY: specify a value to quantify a measure; and

TEXT: enter a text string as data.

The model of interactive applications as graphical editors mainly concerns itself with three tasks: SELECT, POSITION, and TEXT. The user interface model therefore defines **interaction task objects** to handle the select, position and text tasks.

We can identify three basic input techniques which provide the basis for describing the three interaction tasks.

[1] **Coordinate-Input:** this technique allows the specification of an (x,y) coordinate. It is used to define both the SELECT and POSITION tasks.

[2] **String-Input:** this technique allows a text string to be input, and is used to define the SELECT and TEXT tasks.

[3] **Function-Key-Input:** this technique allows a function key to be associated with a particular choice and is used to define the SELECT task.

The user interface model defines a corresponding set of **input technique objects** to handle coordinate, string and function-key inputs.

Each of these input techniques can be realized by many physical input devices. As described in [84], input devices can be classified into five logical categories: picks (e.g., lightpen), locators (e.g., tablet, mouse, trackball, joystick, touch tablet, sonic tablet), valuators (e.g., potentiometer), keyboard, and buttons (e.g., programmed function keyboard, chord keyboard). Note that the examples provided for each category are currently available physical input devices that are natural to the category. However, most computer systems available today provide only a keyboard with function keys and a locator such as a mouse. It is necessary to have the capabilities of all logical categories to provide a rich mix of parameter specification possibilities. It is possible, however, to simulate the logical function of any category with any input device. Some of these simulations are extremely awkward and can therefore be ignored (for example consider simulating a keyboard). Our design model therefore allows any physical input device to be used for the three input techniques outlined above. The physical input device is mainly concerned with accepting input signals from the user and returning these signals for processing by the user interface. The model defines **action table objects** that are used to translate these input signals into meaningful internal actions that can be operated on by the appropriate input technique object. As an example, the actions associated with the Coordinate-Input technique are UP, DOWN, LEFT, RIGHT and ENTER. The first four actions correspond to cursor movement, while the ENTER action designates the current (x,y) coordinate as the parameter which is passed to the appropriate interaction task object to process. If a joystick is used as the input device, then the signals corresponding to joystick movement would be translated by an appropriate action table object to the corresponding LEFT, RIGHT, UP, or DOWN action. The signal that is returned by the joystick when the button is pressed will be translated by the action table object to the ENTER action. If a keyboard is used as the input device then certain key signals (for example ↑ <--, -->, ↓ would be translated as cursor movement actions while a designated key (e.g., ENTER or RETURN) signal would be translated by the action table object to the ENTER action. Note that allowing the user to customize action table objects greatly personalizes parameter specification.

To complete the specification of interaction task objects, the input value returned by the input technique object needs to be mapped to appropriate internal values. Note that the input technique object returns a value in terms of the specified user interface. These values need to be transformed into internal values that can be operated on by the application. **Mapper objects** transform input values from input technique objects into entities that can be operated on by interaction task objects. Interaction task objects in turn pass these parameter values to the appropriate service interface object. Figure 3.12 depicts the overall model for interaction showing the data flow through the model. Figure 3.13 shows the associations of the tasks, mappers, input techniques, action tables and input devices supported by the model.

An important measure of the effectiveness of a model for user interfaces is the speed and ease with which new interaction tasks, input techniques and interaction devices can be incorporated. The model for parameter specification presented above is very flexible, and allows new interaction tasks, input techniques and interaction devices to be easily added. The inclusion of a new interaction task necessitates the definition of a new interaction task object and a corresponding mapper object that provides the facilities to map all possible inputs acceptable by the task. The inclusion of a new input technique necessitates the definition of a new input technique object (and possibly a set of corresponding action table objects), and the definition of a new mapping function in each of the mapper objects that can process the input. Finally, new interaction devices necessitate only the definition of appropriate action table objects.

3.2.3. The Overall User Interface Model

To summarize, the user interface model comprises the following six objects.

[1] **Function Interaction Objects:** These objects control the interaction between the user and a particular application provided function. They control their associated service interface objects, which perform the actual interaction with the user.

[2] **Service Interface Objects:** These objects are in charge of the actual interaction with the user. Specifically, these objects receive a parameter from the user through associated interaction task objects, call the associated service within the application, and display the results of the service to the user through display objects.

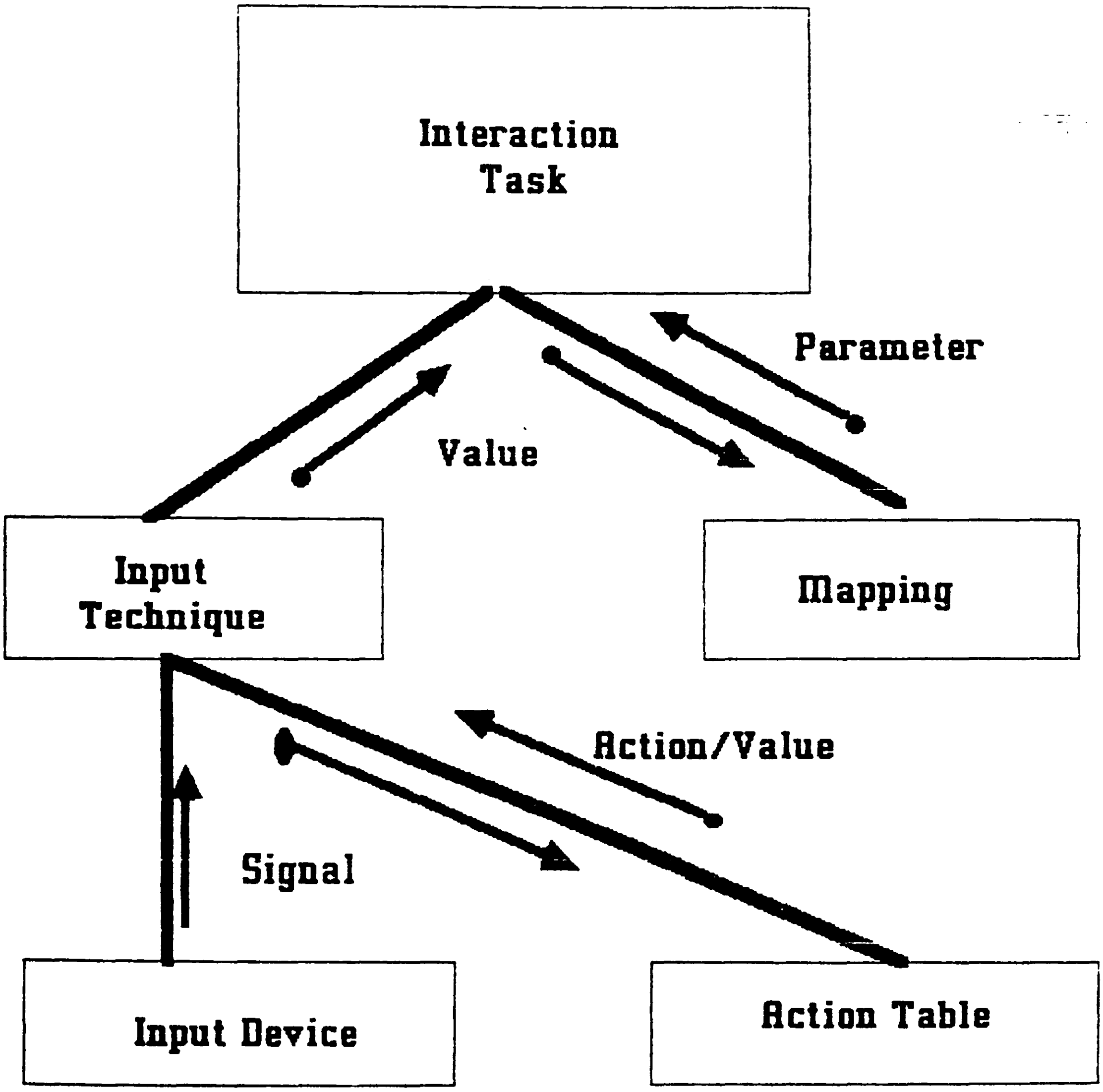


Figure 3.12: Data Flow for Parameter Specification

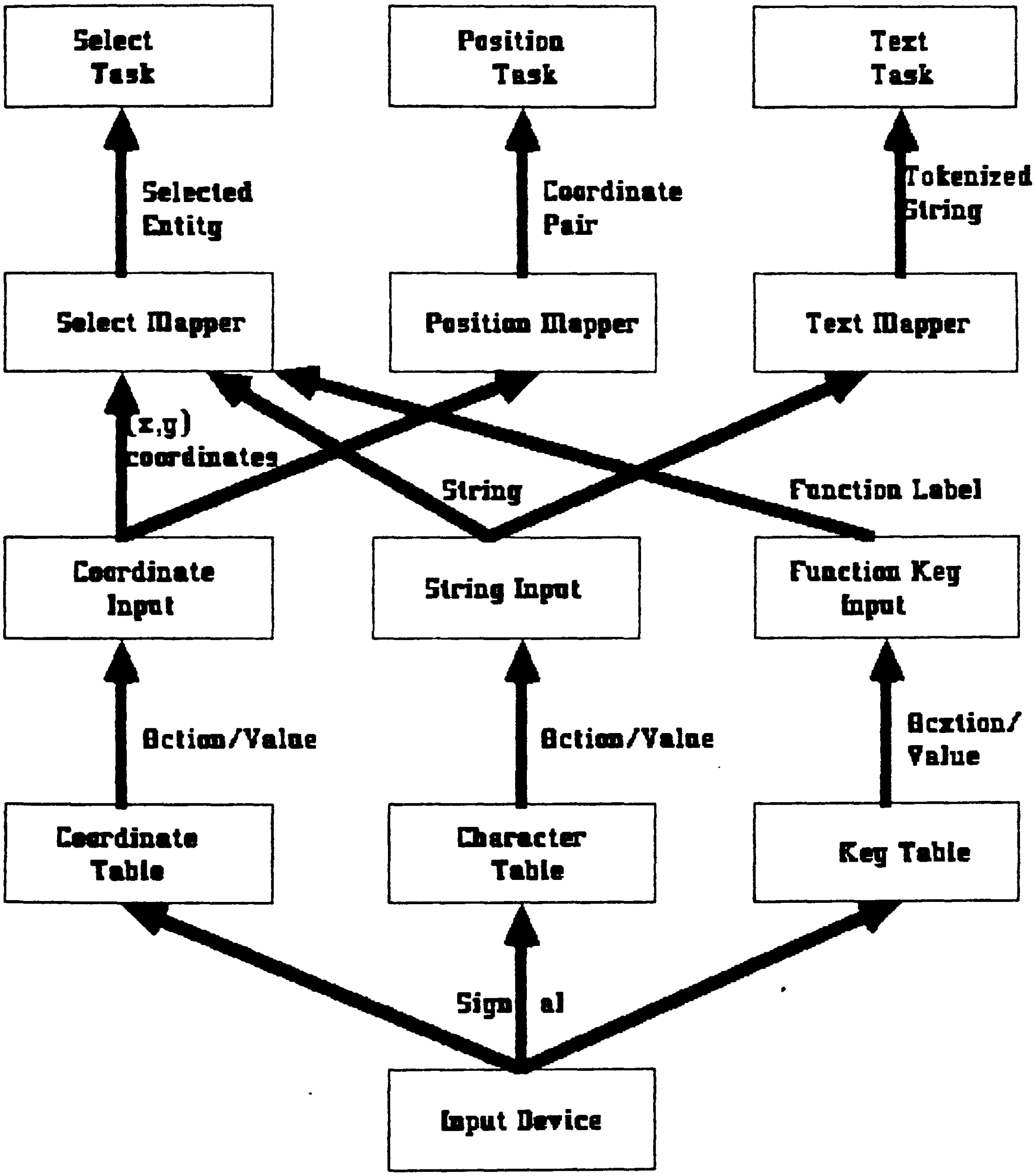


Figure 3.13: Relationships of Parameter Specification Objects

[3] **Name-Mapping Objects:** These objects provide a one-to-one mapping between user defined external representations and internal representations of any entity involved in interaction. These objects facilitate the translation of parameters from user defined representations to application specific representations, and also any internal entities involved in the information to be presented to the user.

[4] **Interaction Task Objects:** These objects facilitate parameter specification by the user. Each interaction task object comprises a **mapper object** to translate all acceptable inputs into forms operable on by the task object. Mapper objects in turn accept input from **input technique objects.** Input technique objects control the interaction (input) devices that are used for parameter specification. **Action table objects** facilitate the translation of signals from these physical input devices into actions or values that are meaningful to the particular input technique object.

[5] **Display Objects:** These objects are in charge of presenting information to the user. This information is usually in the form of internal display buffers maintained by the user interface.

[6] **Device Objects:** These objects are the physical input and output devices that are part of the interactive system.

It is important to note that the user interface model also includes interactive data structures to model the internal structures maintained by the user interface to aid in interaction, for example, menus.

Figure 3.14 is an operational snapshot of a typical interactive system depicting the major objects that are involved in both the application and user interface components. An interaction cycle is described as follows. The user uses the input devices to enter information into the system. User input is managed by interaction task objects. The final input value is transformed by name mapping objects into an internal value that is passed to the appropriate service interface object. The service interface object invokes the appropriate service within the application

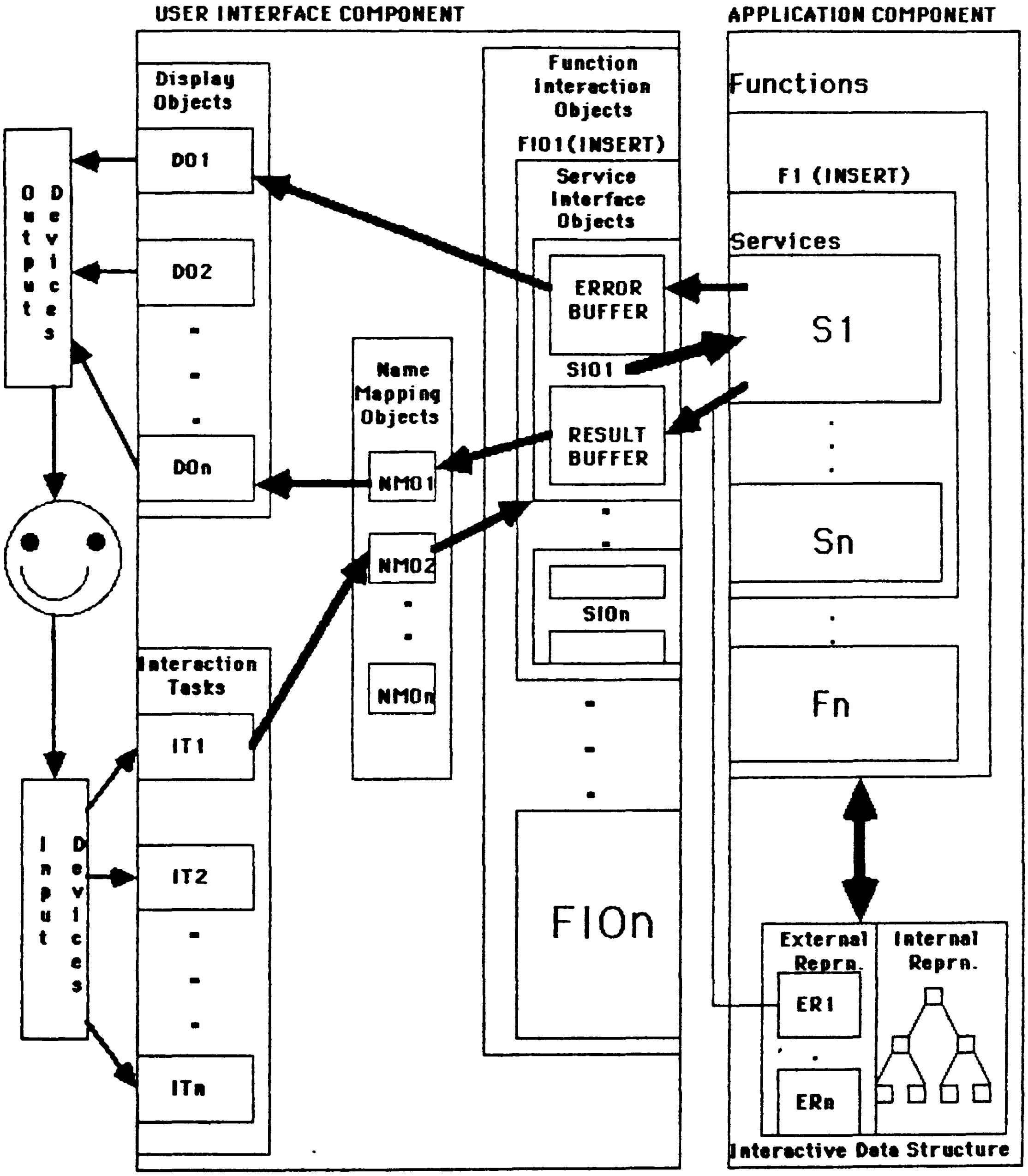


Figure 3.14: Operational Snapshot of the Interactive System

component, passing it the appropriate user input. The results of the service are first transformed by name mapping objects into external user defined representations, and then passed to display objects. Display objects present the information to the user on a physical output device.

It is important to note that the user interface component of interactive systems also provides a set of functions to support interaction. These high-level user interface functions provide facilities for screen layout (such as defining, moving, resizing and scrolling viewports, resizing windows, etc.), manipulating menus, setting global interaction attributes (such as color, screen size, etc.), and invoking function interaction objects. These user interface functions are modeled exactly as application functions, thereby necessitating function interaction and service interface objects to be defined. The user is provided similar facilities to customize every aspect of interaction with these user interface functions. Thus, the MIMD model facilitates a total customization of every aspect of interaction within any interactive software system.

CHAPTER 4

THE DESIGN OF THE MIMD USER INTERFACE DEVELOPMENT ENVIRONMENT

This chapter transforms the MIMD model developed in the previous chapter into a concrete design. The presentation of the design is organized as follows. The first subsection presents an overview of the user interface generation process, and describes the information that is needed from the application component to form the basis of a specification mechanism for describing user interfaces. The next subsection introduces the object-oriented philosophy of system design and implementation. The design of the MIMD UIDE, using the object-oriented paradigm, is then presented in the following two subsections. The first of these subsections deals with the design of the objects involved in presenting information to the user. The second concentrates on the design of objects involved in function invocation and parameter specification.

4.1. The User Interface Generation Process

Figure 4.1 depicts the process of generating a user interface using the MIMD approach. This process is comprised of two phases, **specification** and **generation.** The user specifies personalized interfaces for a particular application in the specificaton phase. This interface specification is used to build the intended interface in the generation phase.

To facilitate the specification of personalized interfaces, the application component provides application specific information. This information is used to define a specification mechanism for creating user interfaces that are appropriate only for the particular application. Users then employ this specification mechanism to define personalized, application specific interfaces. The next subsection describes the details of this application specific information.

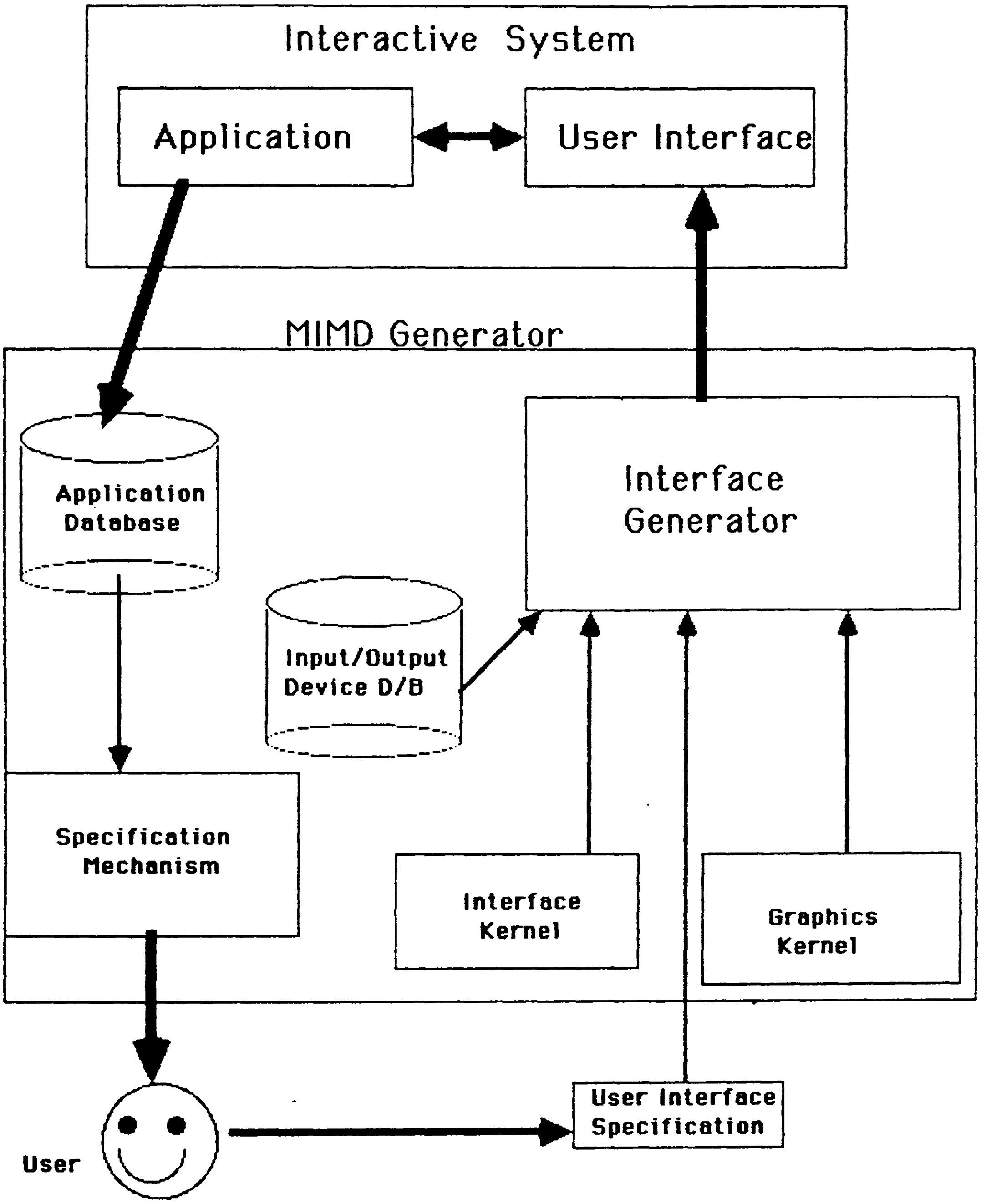


Figure 4.1: The User Interface Generation Process

The development of user interfaces requires, at a minimum, the following components:

[a] a device-independent **graphics kernel** that provides support for the presentation of information to be interfaced to the user,

[b] an **interface kernel** that defines the data structures and routines common to all interfaces, and

[c] an **input/output device database** that describes the characteristics of all input/output devices supported by the underlying operating environment.

The generation phase merges the user interface specification with the above components to generate a personalized user interface for some particular application.

4.1.1. Application Information for the Specification Mechanism

The first issue to be considered in designing MIMD UIDEs is the information that must be provided by an application to form the basis for developing an appropriate specification mechanism. This information consists of three components: information describing the entities of the application, information describing the external representations of the major data structures, and information describing the functions provided by the application.

As was pointed out in the previous chapter, the base entities of an application are dependent on whether the application manipulates a structured or unstructured picture. Throughout our model, we will assume a structured picture since unstructured pictures are just a special case. The possible forms that a structured picture can take are appropriately defined by a two-dimensional context-free language [85, 86]. The symbols, that is, the terminals and nonterminals of the underlying grammar define the base objects of the structured picture, while the productions of the grammar, define the manner in which base objects can be combined. The entity information provided by the application is appropriately maintained in a **language table.** The language table contains the following data for each entity of the language:

[a] the internal name of the entity;

[b] the type of the entity; that is, terminal, nonterminal or production;

[c] the display routine for the entity;

[d] a list of applicable productions for nonterminal entities; and

[e] a description of the entity to aid the user in understanding its role within the application. (For example, a production's role is to define how its left-hand side nonterminal is expanded. It is therefore presented to the user as a possible substitution for the nonterminal entity. During interaction, nonterminal entities present their associated productions as selections for expansion.)

The language table serves two purposes. It allows the user to understand the structure of the picture that is manipulated by the editor. It also provides the necessary information for the specification mechanism to facilitate user defined customizations of entity names.

The information describing the external representations of the major data structures maintained by the application is defined appropriately in a **view table.** The view table contains the following information for each external representation that is supported:

[a] the identifier of the display buffer that contains the external representation, and

[b] a description of the external representation including an appropriate example of its form.

The information provided by the view table is used by the specification mechanism to allow the user to define customized layouts of the external representations. The view table also facilitates understanding of the manner in which the internal state of the application is presented.

The **function table** contains the relevant information about the functions provided by the application. Each entry in the table contains the following information for each function:

[a] the name of the function;

[b] a **service table** that contains information about the services provided by the function. An entry in the service table contains the following information:

  [1] the name of the service;

  [2] the name(s) of the parameter(s) that must be specified;

  [3] the type of the parameter(s), for example, a language entity, an (x,y) coordinate in an external representation, a function, a service, a boolean, a character, a string, an integer, or a real number;

  [4] the range of possible values for the parameter(s), if applicable;

  [5] the default value(s) for the parameter(s), if applicable;

  [6] information to aid in parameter specification;

  [7] the error feedback generated by the service for invalid parameter specification; and

  [8] the resulting information that the service provides after processing the valid parameter(s).

[c] an ordering of services that define the (default) action (either a service call or quitting the function) that is taken after successfully completing the service (note that the default action for an error is to call the service again); and

[d] documentation on the function, describing each service in detail.

The specification mechanism uses the information provided by the function table to present the functionality of the editor to the user. This information also facilitates the customization of the entire interaction process, ranging from the names of functions and services to the manner in which they are invoked.

Since the main objective of the MIMD UIDE is to allow end users to develop personalized interfaces, ideally the specification mechanism itself is an interactive system. The specification mechanism presents the application related information that is contained in the above tables to the user and allows the user to specify the relevant aspects of the intended user interface. Since this application specific information is always available, customization of the user interface is possible at both the micro (for example, customization of entity names) and macro (for example, customization of interacting with a function) levels. This approach allows the user interface to evolve piece-meal with the user's experience.

4.2. The Object-Oriented Paradigm

The object-oriented philosophy of system design and implementation is being heralded as the **structured programming of the 1980's.** The term object-oriented was first used to describe the Smalltalk programming environment developed at Xerox PARC [79, 87]. Object-oriented programming is currently the **hottest technological fad** in software engineering as is evident from ongoing research in the area [31, 88, 89, 90, 91] and the enormous popularity of conventions and tutorials devoted to this topic. The major reason for its popularity can be attributed to the fact that it is one of the best available software engineering design methodologies that supports both encapsulation and reusability, the bases for modern software engineering.

In this dissertation, the following model and terminology of object-oriented design is assumed. The design of an object-oriented system is described by a set of **classes.** Each class defines all the information necessary to construct and use its particular kind of objects. Each object is therefore an **instance** of one class. Each class defines

[a] how its instances are **created,**

[b] a collection of **instance variables** that describes the local storage for maintaining each instance's individual state, and

[c] a set of operations (or **methods**) that can be performed on instances of the class, that is, methods that are generic and apply to any instance of the class.

A class can be defined in terms of one or more other classes using **inheritance.** If a class B is defined in terms of class A, then B inherits the instance variables and methods of class A. Class A is the **superclass** of class B, and conversely B is the **subclass** of A. Class B can override or restrict any of the methods that it inherits from its parent class A. Of course, class B can also define its own instance variables and methods. Thus an object-oriented system is designed as a hierarchy of classes.

An object-oriented software system is therefore a collection of objects. Communication is achieved through **messages** that invoke one of the set of operations defined on the object. Thus, objects in a object-oriented system are encapsulated modules since they can be accessed only through the set of operations that define their external interface. Classes and inheritance promote reusability since newer classes can be defined as derivations of existing classes. (Cox [31] aptly uses the term **Software-IC's** to describe the predefined classes of any object-oriented system.)

Finally, it is important to note that most object-oriented systems define a class Object which is the root of the inheritance hierarchy of all classes within the system. The Object class is the most generic class, and is the only class that has no superclass. It defines how objects are managed by the system and describes a repertoire of behaviors that will be inherited by all other classes. It also provides the basic protocol for instantiating objects of any class. Since the Object class is dependent on the particular implementation environment of the object-oriented system, we will assume its existence for the purposes of our design without giving particulars about its specification.

The presentation of the design for MIMD UIDEs employs the following framework to define classes. Each class definition describes its parent class, along with the instance variables and the set of methods that it provides. The class definition will not include how instances of the class are created, as these are implementation dependent and best decided during the detailed design and implementation phases.

4.3. Presenting Information to the User

This section develops the design of objects involved with presenting information to the user. The first subsection presents the design of interactive data structures. The following subsection discusses the design of display objects.

4.3.1. Interactive Data Structure Objects

Interactive data structures form the basis for managing and manipulating data within the application and user interface components of interactive systems. The external representations of interactive data structures, which are maintained within internal display buffers, describe the information to be interfaced to the user.

The following subsections develop the design of interactive data structures. The first subsection describes the design of the data component of these interactive data structures, while the second subsection describes the design of the structure component. These two subsections consider interactive data structures that support only a single external representation. The next subsection extends the design of interactive data structures to support multiple external representations. The final subsection describes the use of interactive data structures within the MIMD UIDE.

4.3.1.1. Data Objects

The design of the data component of interactive data structures is defined by three subclasses of Object:

[1] the InternalData class which defines the internal representation of data,

[2] the ExternalData class which defines the external representation of data, and

[3] the InteractiveData class which merges the above classes to define the data component of interactive data structures.

The following subsections present the design of these three data classes.

4.3.1.1.1. Internal Data

The InternalData class is an abstract class that is defined as the root of the internal representation of the data abstraction inheritance hierarchy. Its contribution to the design is not in the power of its instances, but in defining a template that describes the methods and local data that must be provided by any class that is derived from it. In other words, the InternalData class defines the abstract external interface (or protocol) that must be provided by any internal data object. The actual definition of the methods is described at the subclass level. Each of the abstract methods defined at the InternalData class level simply return **subclassResponsibility** which is an error indicating that the subclass was responsible for providing the definition of the method. Figure 4.2 depicts the specification of class InternalData. Note that the instance variable dataInfo is a dummy variable to facilitate the description of the methods. This instance variable will be appropriately redefined at the subclass level.

Class: InternalData
SuperClass: Object

Instance Variables:
[a] Object dataInfo; /* the actual data */

Methods:

```
[1] ASSIGNMENT OPERATIONS:
    assignData(Value); /* Assign Value to dataInfo */

[2] RETRIEVAL OPERATIONS:
    retrieveData(); /* return the value of dataInfo */

[3] COMPARISON OPERATIONS:
    isEqual(anObject); /* return TRUE if anObject is equal to dataInfo,
                                            FALSE otherwise */
[4] ATTRIBUTE OPERATIONS:
    size(); /* return the size of dataInfo */
```

Figure 4.2: Class InternalData

The base data types provided by any object-oriented system will be defined as subclasses of the InternalData class. Thus classes Integer, Real, Character, and String will be derived from class InternalData and will redefine the methods accordingly. Since the specification of these base type classes is straightforward (and is usually provided as part of the object-oriented system), their definitions will be assumed. The following discussion defines class EmployeeData which describes the information maintained for each employee for the example interactive system, EMPSYS. The dataInfo instance variable of class EmployeeData is appropriately described as

```
dataInfo = { String Name;
             Integer SS#;
             Integer Pos;
             String Dept;
             Integer Controls;
             EmployeeData* FirstEmp, LastEmp, NextEmp, Boss;
           }
```

Thus Name and Dept are objects of class String while SS#, Pos (0: CEO, 1: Manager, 2: Employee} and Controls are objects of class Integer. All the other fields signify identifiers of EmployeeData objects. Note that the syntax used throughout this design is based on the C programming language [92]. The methods for the EmployeeData class are described in terms of the methods provided by the classes of its subfields, as follows.

[1] **Retrieval Operations:** We define retrieval methods for each field, as follows.

```
retrieve<F_NAME>(); /* return(<F_NAME>-->retrieveData()); */
```

where <F_NAME> is substituted by the name of each field. Note that the terminology

<object_name>--><method_name>(<parameters>);

is used throughout the design for invoking an object's method.

[2] **Assignment Operations:** The assignData(Value) method assumes that Value is an object of class EmployeeData. Its definition follows.

```
Name-->assignData(Value-->retrieveName());
SS# -->assignData(Value-->retrieveSS#());
Pos -->assignData(Value-->retrievePos());
Dept-->assignData(Value-->retrieveDept());
*...etc...*
```

[3] **Comparison Operations:** Let us assume that two objects of EmployeeData are equal if their social security numbers are equal. Thus the method isEqual(anObject), where anObject is a EmployeeData object, is defined as

```
return(SS#-->isEqual(aObject-->retrieveSS#()))
```

[4] **Attribute Operations:** The size() method of EmployeeData is just the addition of the size of its component parts (Name, SS#, Pos and Dept), and therefore can be described as

```
return(Name-->size() + SS#-->size() + Pos-->size() + Dept-->size())
```

The assumption is that the size() method of any object returns an Integer object, and the '+' method is defined by the Integer class. Thus the expression "3 + 4" is an invocation of the '+' method of Integer object 3 which adds its value to parameter 4 and returns the sum 7 as an Integer object.

The above definition of class EmployeeData depicts the reusability aspect of object-oriented systems. The system need only provide the base type class definitions as appropriate derivations of the abstract class InternalData. The designer of the interactive system reuses these base classes to define the data that is specific to the needs of the interactive system.

4.3.1.1.2. External Data

The definition of the ExternalData class is depicted in Figure 4.3. Note that InternalData and ExternalData are parallel hierarchies, with the Object class as their common parent. The reason for separating these classes is due to the fact that the design of an interactive system will normally require the use of non-interactive data. That is, data that will never be interfaced to the user and therefore does not require a definition of its external representation. The design of the interactive system will create instances of class ExternalData to describe the external image of only those InternalData objects that participate in the external representation of interactive data structures.

As an example, consider the ss#_view external representation of the example interactive system EMPSYS. Each node will declare an object of ExternalData that is capable of generating the node's display in the ss#_view display buffer. The instance variable displayFunction contains the necessary information to convert the Integer object SS# of the internal representation to its appropriate display in the buffer. For example, if SS# has the value 593142618, then the

```
                         Class: ExternalData
                        SuperClass: Object


Instance Variables:
[a] Function displayFunction; /* the display routine */

[b] Object displayAttributes; /* the attributes of the display, i.e.,
                               its width and height */

[c] Object entryCoords; /* the entry coordinates of the node's display
                               in some display buffer */


Methods:
[1] ASSIGNMENT OPERATIONS:
   - assignDisplay(dispFunc, dispAttr); /* sets displayFunction to dispFunc and
                                        displayAttributes to dispAttr*/
   - assignEntry(xyPair);  /* sets entryCoords to xyPair */

[2] RETRIEVAL OPERATIONS:
   - display(dispBuf); /* displays node in buffer dispBuf at coordinates
                         entryCoords by invoking displayFunction */
   - retrieveEntry(); /* returns entryCoords */

[3] COMPARISON OPERATIONS:
   contains(xyPair); /* return TRUE if xyPair lies within the boundaries
                         of the node's display, FALSE otherwise */

[4] ATTRIBUTE OPERATIONS:
   displayAttr(); /* returns displayAttributes */
```

Figure 4.3: Class ExternalData

displayFunction might display it as follows:

593-14-2618

4.3.1.1.3. Interactive Data

To complete the design of the data abstraction of interactive data structures, the external and internal representations must be merged into a single unit. In systems that support multiple inheritance, the class InteractiveData is defined as a child of both the InternalData and ExternalData classes. However, since many of the current object-oriented systems do not support multiple inheritance, a more practical design of class InteractiveData combines the two representations by declaring instance variables of each class. Its methods must be designed to provide access to the internal and external representations respectively. Figure 4.4 depicts the specification of class InteractiveData. Note that class InteractiveData could include methods to expose the methods defined for the InternalData and ExternalData classes. For example, the method **isEqual(anObject)** to compare two InteractiveData objects can be defined as:

return(dataPart-->isEqual(anObject-->accessInternal())).

Class: InteractiveData
SuperClass: Object

Instance Variables:
[a] InternalData dataPart; /* the internal representation */

[b] ExternalData displayPart; /* the external representation */

Methods:
[1] accessInternal(); /* returns dataPart */

[2] accessExternal(); /* returns displayPart */

Figure 4.4: Class InteractiveData

Figure 4.5 summarizes the inheritance hierarchy of the data classes.

4.3.1.2. Structure Objects

The design of the structure component of interactive data structures follows the same principles used in the design of the data component. Specifically, the design must allow for both conventional structures as well as interactive structures. The development of interactive systems will include internal structures that need not be interfaced to the user, and therefore do not require any external representations. To exploit reusability, our design of interactive data structures must take advantage of the internal structures. Thus the design defines two parallel inheritance hierarchies, the structure hierarchy which describes the internal structure abstraction, and the interactive structure hierarchy which extends the internal structure abstraction to include external representations.

4.3.1.2.1. Internal Structures

The Structure class is an abstract class that defines the root of the internal structure inheritance hierarchy. This generic class defines the protocol that describes the behavior of any structure, such as lists, trees, tables, sets, etc. It defines the set of methods that must be provided by any structure class that is derived from it. The implementation of these abstract methods within the Structure class is as **subclassResponsibility.** This produces an error message indicating that the definition of the method is the responsibility of the subclass.

Figure 4.6 summarizes the specification of the Structure class. The following points are worth noting about the specification.

[1] The data contained in each node of the structure is defined as an instance of class Object to allow structures of any data object to be defined. Note too that such a design allows for the definition of both **homogeneous** as well as **heterogeneous** structures. It is obvious that the design allows homogeneous objects of a particular class in a structure. Homogeneity can be achieved as a side effect of inheritance. A structure specified in terms of a

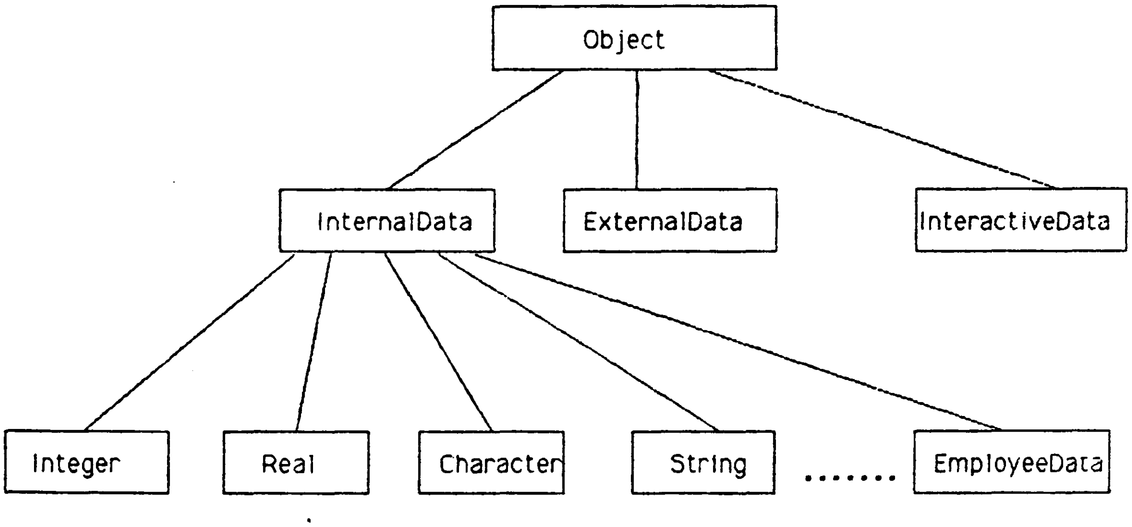


Figure 4.5: The Data Abstraction Hierarchy

Class: Structure
SuperClass: Object

```
Instance Variables:
[a] nodeInfo = { Object nodeData; /* the data in each node of the structure */
                 Links nodeNeighbors; /* a node's neighborhood relationship */
               };

Methods:
[1] ACCESS OPERATIONS:
   - nextNode(aNode, aLink); /* returns aLink neighbor of aNode */
   - eachNodeDo(codeBlock); /* accesses each node in the structure and
                               makes each node perform codeBlock */

[2] SEARCH OPERATIONS:
   searchNode(aObject); /* uses traversal method eachNodeDo to determine
                           if aObject exists in the structure. Returns
                           identifier of node if found, NULL otherwise */

[3] MANIPULATOR OPERATIONS:
   - addNode(aNode, aLink, aObject); /* adds a new node aObject as aLink
                                        neighbor of aNode */
   - deleteNode(aNode); /* deletes node aNode from the structure */
```

Figure 4.6: Class Structure

particular class can hold objects of any class derived from that class; that is, it may be heterogeneous. For example, assume a structure that is defined to maintain objects of class A. Suppose classes B and C are derived from class A. The structure can also hold B and C objects, since their behavior is compatible with A objects.

[2] The iterator method eachNodeDo accepts a block of code, codeBlock, that each node in the structure is asked to perform. The nature of codeBlock is dependent on the particular implementation style of the object-oriented system. Systems like Smalltalk [79] and Objective-C [31] allow an actual block of statements (enclosed between square brackets) to be passed as an argument (or selector of a message). In other systems, codeBlock could be defined as a function that contains the statements that each node will perform. (Refer to [94] for a further discussion on iterators for abstract data structures.) We will assume the former style.

[3] As an example of using the iterator, the method searchNode can be described as follows.

```
codeBlock = [ [eachNode do:
            Boolean isFound;
            isFound = eachNode-->isEqual(aObject);
            if (isFound)
              return(eachNode); /* Node is found */
          ]
          return(NULL); /* Node was not found */
        ];
return(eachNodeDo(codeBlock));
```

The description of specific structures is defined as derivations from class Structure. As with the design of the data component, it is useful to identify a set of basic structures on which other structures can be built. It is widely accepted in computer science education that the list and tree structures are the basic structures. Figure 4.7 presents an appropriate inheritance hierarchy for internal structures which represent most of the major data structures that are employed in designing software systems. Note that even though lists and trees are the base structures, class Set and Graph have parallel inheritance hierarchies. This allows their definitions to be based on any of the base structures.

Figure 4.8 presents the specification of the list structure. Note that nodes within a list are accessed by their position in the list. Thus class List provides additional methods to facilitate processing of nodes based on their position in the list.

The queue structure restricts access to only the first and (currently) last element of the list. Thus class Queue can be simply derived from class List by overriding the following methods.

[1] The access methods are redefined as accessFirst() and accessLast() which provide access to the first and last nodes in the queue. These can easily be defined in terms of the methods provided by List as accessNode(1) and accessNode(numberNodes) respectively.

[2] The addNode method of class Queue adds a node always to the front (or the end) of the queue, and can be described using List's method addElement(1, aObject).

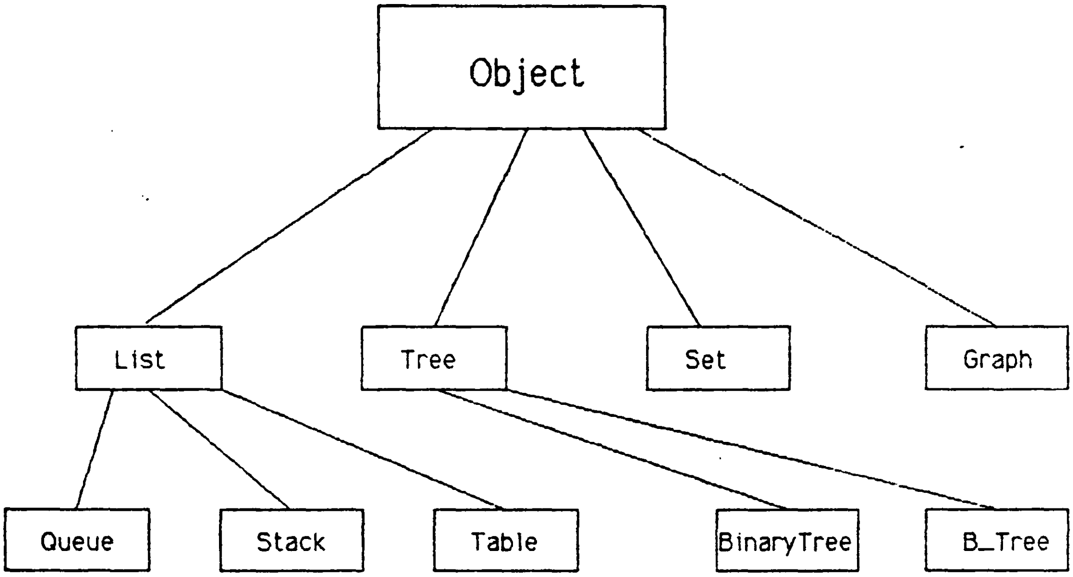


Figure 4.7: The Structure Abstraction Hierarchy

Class: List
SuperClass: Structure

```
Instance Variables:
[a] nodeInfo = { Object nodeData; /* the data in each node of the structure */
                 Link next;      /* each node has a next neighbor */
                 Link previous: /* and a previous neighbor */
               };
[b] Link firstNode; /* access to the first node in the list */
[c] Link lastNode; /* access to the last node in the list */
[d] Integer numberNodes; /* the number of nodes (currently) in the list */


Methods:
[1] ACCESS OPERATIONS:
   - nextNode(aNode, aLink); /* returns next or previous neighbor of aNode */
   - accessNode(aPos); /* returns node at position aPos in the list */
   - eachNodeDo(codeBlock); /* traverse list from firstNode to lastNode */

[2] SEARCH OPERATIONS:
   searchNode(aObject); /* uses traversal method eachNodeDo to determine
                          if aObject exists in the structure. Returns
                          identifier of node if found, NULL otherwise */

[3] MANIPULATOR OPERATIONS:
   - addNode(aNode, aLink, aObject); /* adds a new node aObject as previous
                                       or next neighbor of aNode */
   - addElement(aPos, aObject); /* adds aObject as aPos element of list */
   - deleteNode(aNode); /* deletes node aNode from the structure */
   - deleteElement(aPos); /* deletes node at position aPos in list */
```

Figure 4.8: Class List

[3] Correspondingly the deleteNode method of class Queue always deletes the node at the end (or the front) of the queue, and can be described using List's method deleteElement(numberNodes).

Note that class Queue must restrict access to the other methods provided by List. This is done by overriding each method to return an error message. Through the rest of this discussion we assume that classes invalidate superclass methods that violate their behavior by returning an error message.

A stack is a list with access restricted to one end only, usually the front. Its methods accessFirst, addNode and deleteNode apply to only the first node of the stack. Here again, class Stack is derived as a subclass of List and its new methods are easily described in terms of methods in class List.

Finally, the table structure is a special kind of List wherein access to nodes is provided by the application of a hash function. The three base methods of list that access a node (nextNode, addNode, and deleteNode) are overridden in the specification of class Table so that a hash method of each data object is invoked to determine its position in the list. It is the responsibility of the data objects to provide this hashing method. An abstract hash method is therefore defined at the Object class level which forces the subclasses derived from it to provide their own definitions.

Figure 4.9 depicts the specification of the BinaryTree class, as binary trees are the most common tree structures used in the development of software systems. The description of the methods provided are self explanatory.

Finally, we close this section with a discussion of class Set. A set can be based on any of the structures we have discussed, as its only requirement is that it cannot contain any duplicate elements. The Set class declares an instance variable SetStruct to be an object of class Structure, thereby allowing a set to be based on a list, table, or tree structure. To ensure that no duplicate elements are added to the set, the Set class defines its own addElement(aObject) method. This is defined in terms of the methods provided by the structure SetStruct and can be accomplished as follows:

```
if (SetStruct-->searchNode(aObject) == TRUE)
  return(error("Node aObject already exists, cannot add"));
else
  (SetStruct-->addNode(aObject));
```

The Set class will also define methods that are specific to sets, such as set union, difference and intersection.

Class: BinaryTree
SuperClass: Tree

Instance Variables:

```
[a] nodeInfo = { Object nodeData; /* the data in each node of the structure */
                 Link parent; /* link to parent */
                 Link leftChild /* link to left child */
                 Link rightChild /* link to right child */
               };
[b] Link root; /* the root of the binary tree */
```

Methods:

```
[1] ACCESS OPERATIONS:
   - nextNode(aNode, aLink); /* returns parent, left or right child of aNode*/
   - eachNodeDo(codeBlock); /* an inorder traversal of the binary tree */
   - eachNodeDoPre(codeBlock); /* a preorder traversal of the binary tree */
   - eachNodeDoPost(codeBlock); /* a postorder traversal of the binary tree */

[2] SEARCH OPERATIONS:
   - searchNode(aObject); /* uses one of the traversal methods above to find
                         aObject. Returns identifier of the node if found,
                                                   NULL otherwise*/
   - isLeaf(aNode); /* returns TRUE if aNode is a leaf node (no left or
                                    right child), FALSE otherwise */

[3] MANIPULATOR OPERATIONS:
   - addNode(aNode, aLink, aObject); /* adds a new node aObject as left or
                                           right child of aNode */
   - deleteChild(aNode, aLink); /* deletes left or right child of aNode */
   - deleteNode(aNode); /* deletes aNode (must be a leaf node) from tree */
```

Figure 4.9: Class BinaryTree

4.3.1.2.2. Interactive Structures

Class InteractiveStruct extends the Structure class by providing facilities to maintain and manage the external representation. The main assumption is that objects of the Structure class have as their node's data, objects of the InteractiveData class. This ensures that data objects include both their internal and external representations.

The InteractiveStruct class is specified in Figure 4.10. The following points serve as an explanation of the specification of the InteractiveStruct class.

Class: InteractiveStruct
SuperClass: Object

Instance Variables:

```
[a] Structure internalStruct; /* the internal representation of interactive
                                        data structures */
[b] DisplayBuffer dispBuf; /* the buffer wherein the external representation
                               of the data structure is generated */
[c] Function displayGeometry(); /* a function that calculates the entry
                           coordinates of nodes' displays in dispBuf */
```

Methods:

```
[1] ACCESS OPERATIONS:
   - nextDisplay(aNode, aLink); /* returns entry coordinates of the display
                                    of aLink neighbor of aNode */
   - eachDisplayDo(codeBlock); /* accesses the external representation of
                           each data node; makes it perform codeBlock*/

[2] SEARCH OPERATIONS:
   searchDisplay(xyPair); /* uses traversal method eachNodeDo to search
                        for a node whose display contains xyPair. Returns
                        identifier of node if found, NULL otherwise */

[3] MANIPULATOR OPERATIONS:
   - addInteractive(aNode, aLink, aObject); /* adds a new node aObject into
                                    the interactive data structure
                                    as aLink neighbor of aNode */
   - deleteInteractive(aNode); /* deletes node aNode from the interactive
                                         data structure */
```

Figure 4.10: Class InteractiveStruct

[1] displayGeometry is a function that is capable of calculating the entry coordinates (within dispBuf) of the display of every node's external image. This function is called whenever the interactive data structure is changed through additions or deletions of nodes. displayGeometry could employ either a brute-force method wherein every node's entry coordinates are recalculated whenever a change is made, or an incremental method wherein the function accepts the identifier of the node that affected the change and recalculates the entry coordinates of only those nodes in the structure that were affected by the change. For simplicity, we will assume the former design of displayGeometry.

The deleteInteractive method is similarly described.

As an example, consider a modification of EMPSYS which maintains only a single external representation, ss#_view. The data structure for EMPSYS is an object of class InteractiveStruct. The instance variable internalStruct defines the list of nodes, dispBuf defines the display buffer ss#_view, and the displayGeometry function contains the necessary information to calculate the entry coordinates of each node's display in dispBuf. The displayGeometry function can be defined in terms of the list's iterator as follows:

```
/* Assume that DWIDTH defines the width of dispBuf */
codeBlock = [ eachNode do:
            /* If 1st node then entry = (0, DWIDTH/2) */
            If (eachNode-->isEqual(internalStruct-->accessNode(1)))
              (eachNode-->accessExternal())-->assignEntry((0,DWIDTH/2));
            Else [
                 /* access previous node */
                 prevNode = (eachNode-->accessInternal())-->retrievePrev();
                 /* extract entry coordinates */
                 xyPair = (prevNode-->accessExternal())-->retrieveEntry();
                 /* adjust y coordinate */
                 xyPair.y += ((prevNode-->accessExternal())-->displayAttr()).height + K;
                 /* assign entry coordinates */
                 (eachNode-->accessExternal())-->assignEntry(xyPair);
                ]
          ]
internalStruct-->eachNodeDo(codeBlock);
```

4.3.1.3. Supporting Multiple External Representations

The following two new classes extend the design of interactive data structures to allow for multiple external representations:

[1] class MultiData, which is derived from class InteractiveData, that associates a list of external representations of data with its single internal representation, and

[2] class MultiStruct, which is derived from class InteractiveStruct, that associates a list of display buffers with a single internal representation to hold its multiple external representations.

Figure 4.11 depicts the specification of class MultiData. The multiple external representations are maintained in variable displayPart which is defined as an object of class List. The accessExternal method provides access to the ith display in the list displayPart, and is suitably described by

```
return(displayPart-->accessNode(i));
```

Class: MultiData
SuperClass: InteractiveData

Instance Variables:
[a] InternalData dataPart; /* inherited from InteractiveData */

[b] List displayPart; /* a list of ExternalData objects, i.e., multiple
external representations of dataPart */

Methods:
[1] accessInternal(); /* return dataPart */

[2] accessExternal(i); /* returns ith external display; i.e., node i of
List displayPart */

Figure 4.11: Class MultiData

Finally, Figure 4.12 shows the specification of the MultiStruct class. Note that the InternalStruct variable is assumed to hold data elements which are objects of class MultiData. The description of methods nextDisplay and eachDisplayDo is similar to the descriptions of these methods for the InteractiveStruct class. The only change is the use of

```
accessExternal(i)
```

to access the ith external representation of the data node. Any manipulation of the data structure must update all of its external representations. Thus, the manipulator methods are redefined in class MultiStruct to change all the external images. As an example, the redisplay of all external representations within the addInteractive method is described as follows.

Class: MultiStruct
SuperClass: InteractiveStruct

Instance Variables:

[a] Structure internalStruct; /* inherited from InteractiveStruct */

[b] List dispBufs; /* a list of display buffers to hold the multiple external representations */

[c] List displayGeometry; /* a list of geometry functions, one for each external representation */

Methods:

[1] ACCESS OPERATIONS:
- nextDisplay(iPos, aNode, aLink); /* returns entry coordinates within iPos display buffer of the display of aLink neighbor of aNode */
- eachDisplayDo(iPos, codeBlock); /* accesses each data node's iPos external representation and makes it perform codeBlock */

[2] SEARCH OPERATIONS:
searchDisplay(iPos, xyPair); /* searches for a node whose iPos external representation contains xyPair */

[3] MANIPULATOR OPERATIONS:
- addMulti(aNode, aLink, aObject); /* adds a new node and updates all its external representations */
- deleteMulti(aNode); /* deletes a node from the structure and updates all its external representations */

Figure 4.12: Class MultiStruct

```
codeBlock = [ Integer i;
            i-->assignData(0); /* i = 0 */
            eachNode do:
              i-->increment1(); /*  i = i + 1 */
              eachDisplayDo(i, [eachDisplay do:
                                eachDisplay-->display(eachNode)]);
          ]
dispBufs-->eachNodeDo(codeBlock);
```

4.3.1.4. Using Interactive Data Structure Objects

The following user interface objects that are involved in the presentation of information to the user necessitate the use of interactive data structures: name-mapping objects, the application specific information that is used to define the user interface specification mechanism, and menus.

Name-mapping objects are used to define mappings between user defined representations and the internal representations of those entities that are interfaced to the user. Thus, name-mapping objects are merely instances of class InteractiveData or MultiData, allowing for both a single representation or multiple representations of internal entities.

The information provided by interactive applications that form the basis for defining an appropriate specification mechanism (the language, view and function tables) are merely instances of class List or Table. The data that describes the contents of each table is defined by an appropriate class which is derived from class InternalData.

User interfaces require the definition of menus to aid in function invocation and parameter specification. Menus maintain names of entities known to the application (function names, service names, and language entity names), which describe possible parameter choices. For personalized user interfaces, menu entries correspond to name mapping objects that describe the user defined representations of internal entities. Menu objects are appropriately defined as instances of class InteractiveStruct or MultiStruct which maintain name mapping objects in an appropriate structure (usually a list or tree (for hierarchical menus)). Figure 4.13 depicts an example of a user defined menu for function invocation. The menu supports three external representations to allow selection of the function according to its iconic representation, its association with function keys and its textual representation. The Figure also depicts the objects involved in a typical interactive data structure that maintains multiple external representations. User defined name mapping objects correspond to the MultiData objects shown in the Figure. Each MultiData object is made up of an InternalData object that defines the function name, and

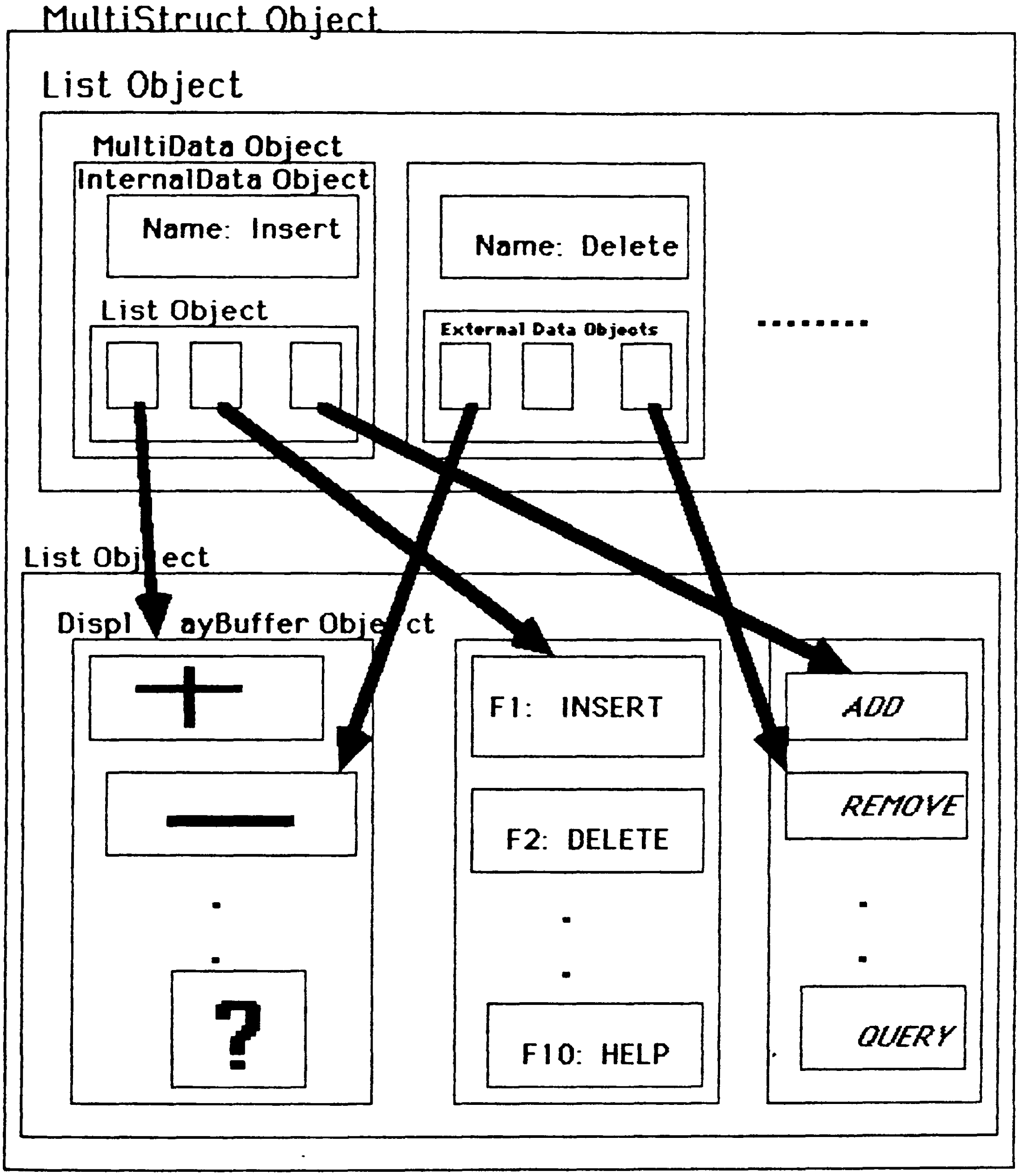


Figure 4.13: Example of a MultiStruct Object

a List object that maintains three ExternalData objects that are capable of generating the appropriate external images of the function name in each of the three display buffers. The Figure highlights the boundaries of the external images generated by these ExternalData objects. The external representations are defined as a List object that maintains three DisplayBuffer objects. Finally, the menu itself is a MultiStruct object that maintains the internal representation along with its three external representations.

4.3.2. Display Objects

Display objects are concerned with interfacing information internal to the interactive system to the user. The information to be interfaced to the user is maintained as suitable external representations within display buffers. These external representations are displayed on appropriate physical output devices for viewing by the user. The transformation of the image within the display buffer to an image on the output device is achieved through **windows** and **viewports.** Windows define rectangular regions within display buffers describing the extent of the image to be interfaced to the user. Viewports define rectangular regions on the physical output device wherein a portion (according to the viewport dimension) of the image defined by the window is depicted to the user.

The design of display objects is defined by the following classes.

[1] Class DisplayBuffer defines the characteristics of internal display buffers, providing the facilities to generate and maintain external images.

[2] Class DisplayDevice defines the characteristics of the final display providing the facilities to maintain and manipulate viewports.

[3] Class View defines the transformation of images from display buffers to output devices through windows and viewports.

The following subsections describe the design of the three classes.

4.3.2.1. Display Buffer Objects

It is appropriate to maintain the image within internal display buffers as a display file of graphical display commands, for the following reasons.

[a] This approach allows any image, however large, to be maintained.

[b] The image maintained within the display buffer must be transformed from world coordinates to device coordinates before being presented on the output device. It is more efficient to apply this transformation on display commands rather than on the actual image itself.

[c] The actual image to be displayed is generated once before being presented to the user.

[d] Changes to the image are accomplished by manipulating (that is, deleting and inserting) display commands. This approach is more efficient than erasing and redrawing images.

[e] Most of the graphics kernels currently available provide image generation facilities only at the viewport level.

Thus, the major concern of the DisplayBuffer class is to provide facilities for manipulating the display file of graphics commands and for extracting an appropriate portion of the image as defined by windows. The specification of class DisplayBuffer is depicted in Figure 4.14. Note that the graphics display commands supported will correspond to similar facilities provided by any graphics standard such as Core [74]. We have included only a few of the possible commands in Figure 4.14. The getWindow method will only return those display commands that are involved in generating the image within the window specification. In other words, getWindow clips the image to the specified window.

4.3.2.2. Display Device Objects

Each display device can have many viewports associated with it. All of these viewports may not be visible at the same time. Therefore display devices provide the facilities to maintain these viewports and their displays. The specification of the DisplayDevice class is shown in

Class: DisplayBuffer
SuperClass: Object

Instance Variables:

[a] List displayCommands; /* the display file of graphics commands defining the external image */

[b] XyCoords curXy; /* the current (x,y) coordinates updated by every display command */

[c] Integer numEnt; /* the number of entries in displayCommand */

Methods:

(** GRAPHICS PRIMITIVES ****)**

[1] moveAbs(xyPair); /* set curXy to xyPair */

[2] moveRel(xyPair); /* add xyPair to curXy */

[3] lineAbs(xyPair); /* generate line command from curXy to xyPair */

[4] lineRel(xyPair); /* generate line command from curXy to (curXy+xyPair) */

[5] circle(xyPair,radius); /* generate circle command */

[6] text(aString); /* generate aString text command */

(** DISPLAY FILE MANIPULATION ****)**

[7] clearDisplay(); /* set numEnt to zero */

[8] deleteDisplay(aPos); /* deletes aPos entry from displayCommands */

[9] addDisplay(aCommand,aPos); /* adds display command aCommand at aPos position in displayCommands */

(** WINDOW INTERFACE ****)**

[10] getWindow(winEntry,winDim); /* return list of display commands corresponding to window specification */

[11] getCurXy(); /* return curXy */

Figure 4.14: Class DisplayBuffer

Figure 4.15. Viewports are maintained in viewStruct which contains the following information for each viewport: the identifier of the viewport object, the dimensions of the viewport, a flag indicating whether the viewport is visible or invisible, and the position of the viewport on the display device. Note that the DisplayDevice class defines display devices at a logical level. The actual association with a physical output device supported by the underlying operating environment is achieved by the physicalDev instance variable. Such a design strategy allows separate instances of DisplayDevice to handle different types of physical output devices. It also makes it easy to handle different emerging technologies by merely deriving subclasses from DisplayDevice to describe the characteristics of the newer technologies.

```
                         Class: DisplayDevice
                        SuperClass: Object

Instance Variables:
[a] String devName; /* the internal name of the display device */
[b] XyCoords devDim; /* the lower right coordinates of the device, assuming
                        that (0,0) is upper right coordinates */
[c] List viewStruct; /* a list of viewports maintained by the device */
[d] ObjId physicalDev; /* the identifier of the physical output device */

Methods:
[1]  identifyView(xyPair); /* returns the top most viewport within whose
                              display xyPair lies */
[2]  clearArea(aView); /* clears the area of the display device affected by
                          the display of aView */
[3]  clearDevice(); /* erases the entire display */
[4]  displayView(aView); /* display aView viewport */
[5]  displayAll(); /* displays all viewports */
[6]  viewInvisible(aView); /* makes aView invisible and erases its display */
[7]  viewVisible(aView); /* makes aView visible and draws its display */
[8]  addView(aView,aPos); /* adds aView at position aPos in viewStruct */
[9]  deleteView(aView); /* deletes aView from viewStruct */
[10] circulate(aView); /* brings aView's display to the top */
```

Figure 4.15: Class DisplayDevice

Viewports must be allowed to **overlap** to facilitate multiple viewports to be displayed simultaneously and to allow the space allocated to viewports to intersect. To facilitate overlapping viewports, it is necessary to define an ordering of their displays. This ordering is defined as their position in the List object viewStruct. Thus, if the space occupied by two viewports A and B intersect, and the position of A in the list is less than that of B, then the display of B will hide the intersection portion of A's display. Figure 4.16 depicts four overlapping viewports and their corresponding position in list viewStruct. In the Figure viewA is the bottom most display while viewD is the topmost display. The following serve as an explanation of the methods provided by class DisplayDevice.

[1] The identifyView method determines a viewport given an (x,y) coordinate on the device. This is done by a reverse traversal of viewports maintained in viewStruct, starting at the top most viewport. The viewport method identifyXy is called to determine whether xyPair lies within its display. This method returns the identifier of the viewport if found, NULL otherwise.

[2] The clearArea method is used to clear the portion of the device that is affected by aView's display. This method returns the identifier of the bottom most viewport whose display was affected. Note that redisplay starts from this viewport.

[3] The clearDevice method clears the entire display space.

[4] The displayView method causes the display of viewports starting with aView and terminating with the top most viewport. This method is described as follows.

```
/* determine position of aView in viewStruct */
startPos = veiwStruct-->searchNode(aView);

codeBlock = [eachNode do:
                eachNode-->displayImage();
          ]

/* call iterator method of viewStruct */
viewStruct-->eachNodeDo(codeBlock, startPos);
```

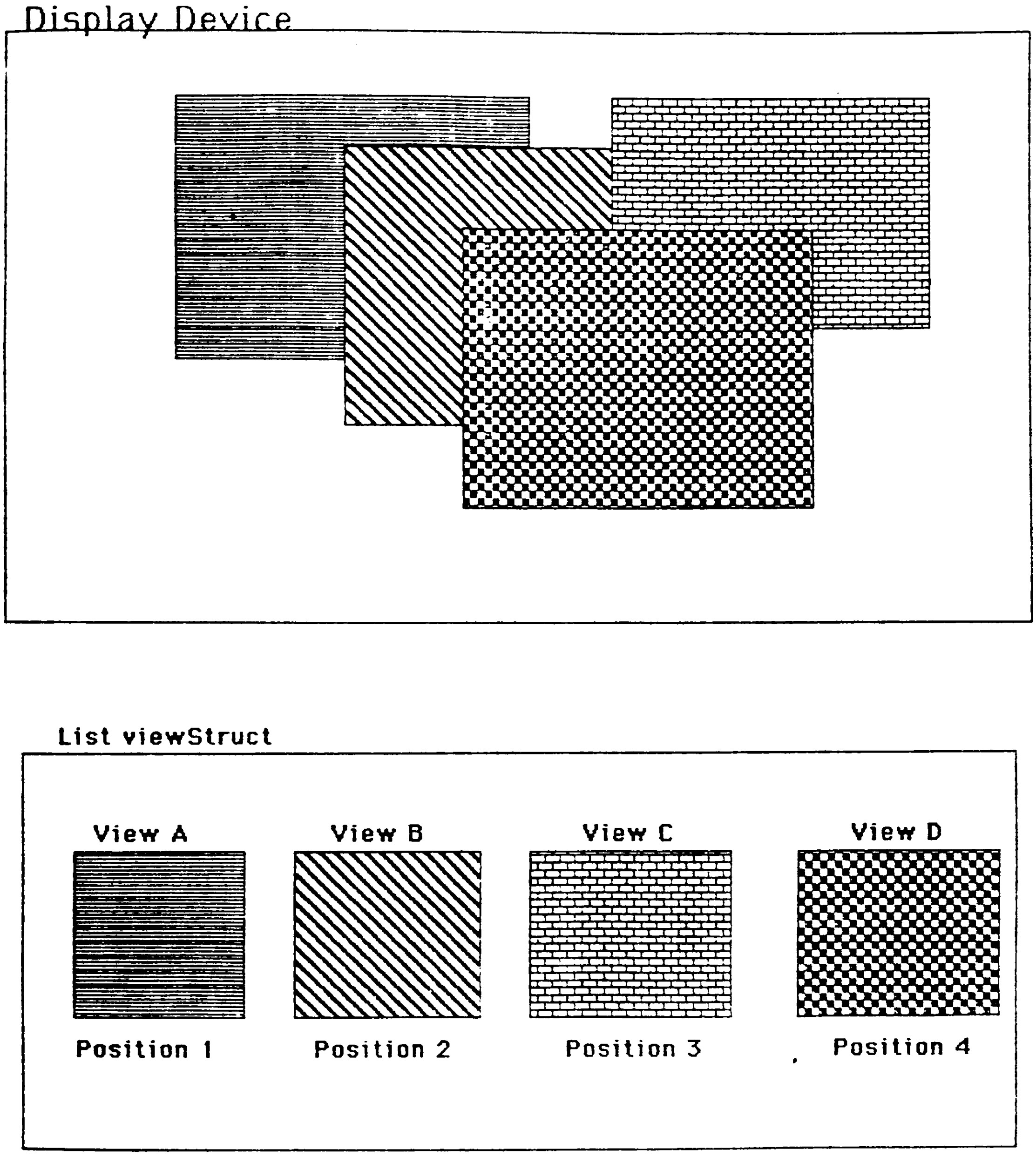


Figure 4.16: Overlapping Viewports

where searchNode and eachNodeDo are methods provided by class List.

[5] The displayAll method causes the display of all viewports. This is done in a similar fashion to displayView, setting startPos to 1.

[6] The viewInvisible method is used to erase a viewport's display, and is described by the following.

```
/* clear area affected by aView */
bottomView = clearArea(aView);
/* set aView's visible flag to FALSE */
aView-->assignVisible(FALSE);
/* redisplay affected area */
displayView(bottomView);
```

[7] The viewVisible method makes a viewport visible and causes its display to appear on the output device. This method is similar to the viewInvisible method, except that the visible flag is set to TRUE.

[8] The addView method allows a viewport to be associated with an output device. The relevant information of the viewport is described in aView and the viewport is inserted at position aPos in viewStruct. Note that if the viewport is visible, then it must be displayed on the output device. The following describes the method.

```
/* add aView to viewStruct */
viewStruct-->addElement(aPos,aView);
/* if viewport is visible, display it */
If (aView-->retrieveVisible() == TRUE)
   viewVisible(aView);
```

[9] The deleteView method causes a viewport to be disassociated from the output device. Note that if the viewport was visible, then a redisplay is necessary. Therefore, the method is described by the following.

```
/* If visible, redisplay */
If (aView-->retrieveVisible() == TRUE)
   viewInvisible(aView);
/* delete from viewStruct */
viewStruct-->deleteNode(aView);
```

[10] The circulate method allows a viewport's display to be the top most display on the output device. This necessitates both a shift from its current position in viewStruct to the top most viewport, and a redisplay. The method is described by the following.

```
/* clear area affected by viewport's display */
bottomView = clearArea(aView);
/* delete viewport from current position */
viewStruct-->deleteNode(aView);
/* add viewport to last position */
viewStruct-->addElement(aView,last);
/* redisplay */
displayView(bottomView);
```

The effect of the circulate method is depicted in Figure 4.17 which shows the result of making viewB of Figure 4.16 the top most display.

4.3.2.3. View Objects

The View class defines the characteristics of viewports and their corresponding windows. Its specification is depicted in Figure 4.18. The relationships between display buffers, windows, viewports, and output devices in interfacing internal information to the user is depicted pictorially in Figure 3.10 (presented in the previous chapter). Note that windows are defined by their entry coordinates within the display buffer and their corresponding width and height. The description of viewports necessitate two entry coordinates, one into the associated window and the other in the display device. Note that scrolling is achieved by moving the window relative to the virtual image defined within the display buffer. The new window is mapped onto the viewport for redisplay. The display of the viewport on the display device is made up of the viewport display, which produces the boundaries of the viewport, and the image display.

The resizeWin method allows the redefinition of the window. Note that both the entry coordinates of the window within the display buffer and the dimensions of the window can be changed. As was pointed out earlier, resizing of windows is useful in obtaining the effects of zooming. Panning or scrolling within the image is achieved by changing the entry coordinates of the window in the display buffer, as defined by the moveView method. The resizeView method

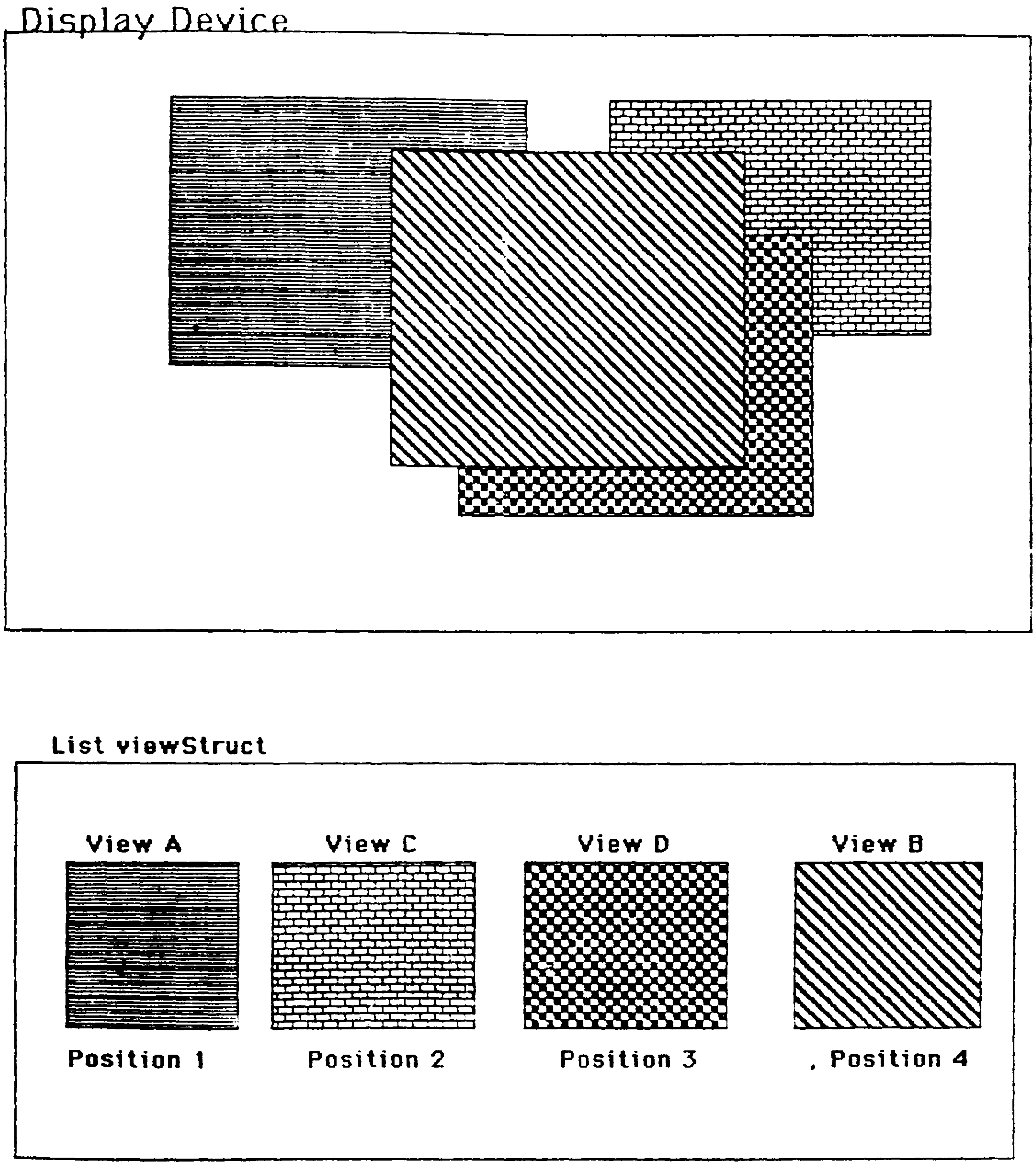


Figure 4.17: Effect of Circulating View B

Class: View
SuperClass: Object

Instance Variables:

[a] String viewName; /* the internal name of the viewport */
[b] Dimension viewDim; /* the width and height of the viewport */
[c] Dimension winDim; /* the dimension of the window */
[d] XyCoords winEntry; /* the entry coordinates of the window in the
display buffer*/
[e] XyCoords viewEntry; /* the entry coordinates of the viewport in the
window */
[f] XyCoords devEntry; /* the entry coordinates of the viewport in the
display device */
[g] Function dispView; /* the display routine that generates the viewport
boundaries */
[h] Object viewAttrs; /* the display attributes of the viewport used by
dispView */
[i] ObjId bufId; /* the identifier of the associated display buffer */
[j] ObjId displayDev; /* the identifier of the associated display device */

Methods:

[1] resizeWin(xyPair,newDim); /* sets winEntry to xyPair and winDim to newDim*/
[2] moveView(xyPair); /* set viewEntry to xyPair */
[3] resizeView(newDim); /* sets viewDim to newDim */
[4] displayImage(); /* generate viewport image on displayDev */
[5] identifyXy(xyPair); /* determines whether xyPair lies within the
viewport's display */
[6] viewToWin(xyPair); /* transforms xyPair from device coordinates to
world coordinates */

Figure 4.18: Class View

allows the dimensions of the viewport to be changed. The displayImage method causes the display of the internal image on the display device. This is achieved as follows. First bufId's method getWindow(winEntry, winDim) is called to extract the display commands corresponding to the image defined by the window. This image is clipped to the viewport specification and transformed from world coordinates to device coordinates to generate the appropriate image on the display device. The other methods are self explanatory.

Let us close this subsection by discussing how a user (x,y) coordinate input is translated into an internal object. The input coordinates are passed to the controlling DisplayDevice object's identifyView method which returns the View object in charge of the (x,y) coordinates. The View object's viewToWin method is called to transform the coordinates from device coordinates to world coordinates. Finally, the InteractiveStruct object that defines the interactive data structure which controls the display buffer is passed the world coordinates to determine the internal object within whose display the coordinates lie.

4.4. Function Invocation and Parameter Specification

The user interface component's main role is to control user interaction with the functions provided by the interactive software system. Therefore, the majcr and most crucial part of personalizing user interfaces is the specification of how the user interacts with the system's functions. User interaction is described as function invocation, parameter specification, displaying results and the specification of the next action to be performed. The previous subsection presented the design of the classes concerned with displaying information to be interfaced to the user. This subsection defines the classes that control function invocation, parameter specification and action specification.

To reiterate, the underlying model for function interaction defines application functions as a set of services that can process their parameters independently. To simplify the design without loss of generality, each service handles the specification of exactly one parameter. On invocation, each service accepts the parameter value, an error buffer and a result buffer. These buffers are merely instances of class DisplayBuffer as presented in the previous subsection. On completion, the service returns the result of processing the parameter in one of the buffers and designates the next action to be performed. The contents of the returned buffer are presented to the user. The specification of the next action to be performed is either a service provided by the current function or quitting the function altogether. Control is therefore passed to the specified service or the user interface's main interaction function which controls the entire interaction process. The above interaction step is repeated by allowing the user to invoke a function provided by the main interaction function until the user quits the interaction function itself, thereby terminating the session with the interactive system.

Service interaction needs an interaction task that will be used for parameter specification. Interactive software systems are mainly concerned with three interaction tasks, selecting a choice form a set of alternatives, designating an (x,y) coordinate in a specified display buffer, and entering text. The user interface specification allows the personalization of parameter specification by describing how user input is translated into an appropriate parameter value required by the interaction task. The model for parameter specification is presented in Figure 3.13 (in the previous chapter). The user specifies the physical **input device** that is used to specify the parameter. The input signals returned by the physical input device are translated into internal actions or values by appropriately specified **action tables.** The action/value is processed by an appropriate **input technique** which controls the input process. When the parameter has been input, its value is passed to a specified **mapper.** The mapper transforms user defined values to internal values. The transformed parameter value is then passed to the appropriate service by the interaction task.

Therefore, function interaction is described by input device objects, action table objects, input technique objects, mapper objects, interaction task objects, service interface objects and function interaction objects. The following subsections present the object-oriented design of these components.

4.4.1. Input Device Objects

The InputDevice class is mainly concerned with the physical aspects of user input; waiting for user input, sensing the input signal, and returning the input signal. The logical classification of input devices into valuators, picks, locators, keyboard and buttons is defined at higher levels by action table objects and input technique objects.

The specification of the InputDevice class is depicted in Figure 4.19. The methods provide facilities to activate and deactivate the input device, as well as to get the signal input from the user. The getInput method waits for the user input and returns the signal to the caller.

Class: InputDevice
SuperClass: Device

Instance Variables:
[a] String inDev; /* the internal name of the input device */

[b] ObjId outDev; /* the associated physical output device */

Methods:
[1] activateDev(); /* activates the input device */

[2] deactivateDev(); /* deactivates the input device */

[3] getInput(); /* waits for user input and returns the signal */

Figure 4.19: Class InputDevice

4.4.2. Action Table Objects

Action table objects are used to translate signals returned by physical input devices into actions or values operable on by the associated input technique object. The main facilities provided by action table objects are allowing the definition of associations between input signals and the corresponding action/value, as well as searching for an action/value given an input signal. Thus, action tables can be defined as instances of an appropriate data structure provided by the system. Figure 4.7 shows the inheritance hierarchy of the classes that define the data structures supported by the MIMD UIDE. In most cases, action tables would be instances of class List.

Following the model of parameter specification, three types of tables can be identified.

[1] **Coordinate Table:** This table is used to maintain associations related to coordinate input. The actions necessary for coordinate input are LEFT, RIGHT, UP, DOWN and ENTER. The first four actions designate cursor movement, while the ENTER action designates the current cursor coordinates as the desired input value. The input signals that are associated with these actions are dependent on the physical input device and defined by the user.

[2] **Character Table:** This table maintains associations related to string input. As pointed out earlier, the keyboard is the sole input device for string input as all other simulations are awkward. Therefore, the associations define relations between keyboard keys and corresponding actions. The actions necessary for string input are the following:

[a] **MOVE:** to move a character cursor UP, DOWN, LEFT or RIGHT;

[b] **ADD:** to add a character to the string;

[c] **DELETE:** to delete a character from the string; and

[d] **ENTER:** to designate the completion of the string input.

The input signals corresponding to these actions are user defined. The following presents the default associations:

[a] ↑ <--, -->,↓key signals for the MOVE action;

[b] alphabetic, numeric, punctuation and white space key signals for the ADD action;

[c] BACKSPACE and DELETE key signals for the DELETE action; and

[d] RETURN and ENTER key signals for the ENTER action.

Note that the action associated with each signal is a token which is a (type,value) pair that defines the type of the action and the value of the action. Thus, the action returned by the '↑ signal would be (MOVE,UP) while the action returned by the 'a' signal would be (ADD, 'a').

[3] **Key Table:** This table is used to maintain associations related to function key input. Each physical input device designates the number of function keys supported. The associations define relationships between the function key signals and their corresponding actions. The actions will usually correspond to identifiers of internal objects (function objects, service objects, name mapping objects, etc.).

4.4.3. Input Technique Objects

Input technique objects control the input aspect of interaction. Their major concerns are to control the input process and return the designated value to the associated interaction task object. The design strategy defines a class InputTech which is the parent of all the input technique objects supported by the system. The subclasses CoordinateInput, StringInput and FuncKeyInput are defined to handle coordinate, string and function key inputs respectively.

The specification of class InputTech is depicted in Figure 4.20. It describes the instance variables common to all techniques and the inAction method that obtains the user input. Note that curAct is made up of two fields: Type and Value, which designate the type of the action and the corresponding value respectively. The inAction method is described as follows.

Class: InputTech
SuperClass: Object

Instance Variables:

```
[a] ObjId viewId; /* the viewport associated with user input */
[b] ObjId bufId; /* the display buffer associated with viewId */
[c] ObjId outId; /* the associated display device object */
[d] ObjId inId; /* the associated input device object */
[e] ObjId actId; /* the associated action table object */
[f] Signal curIn; /* the current input signal */
[g] Action curAct; /* the action associated with curIn */
```

Methods:

```
[1] inAction(); /* gets user input, sets curAct */

[2] processAction(); /* processes the action in curAct */
```

Figure 4.20: Class InputTech

```
/* get input signal */
curIn = inId-->getInput();
/* search for curIn in action table */
actNode = actId-->searchNode(curIn);
/*retrieve action from actNode, if found */
if (actNode != NULL)
   curAct = actNode-->retrieveAction();
else /* ERROR */
   curAct = ERROR;
```

The processAction method must be provided by any class derived from class InputTech to control the processing of input. It is defined in class InputTech to force every subclass to define the method.

Figure 4.21 depicts the specification of class CoordinateInput. The processAction method is described as follows.

Class: CoordinateInput
SuperClass: InputTech

Instance Variables:
[a] ObjId cursorId; /* the associated cursor object */
[b] Dimension curDim; /* the dimensions of the cursor */
[c] XyCoords curEnt; /* the entry coordinates of the cursor on the
display device */
[d] String errMsg; /* the message indicating an error in input */
[e] ObjId errBuf; /* the DisplayBuffer object in charge of error messages */
[f] ObjId errView; /* the View object associated with errBuf */

Methods:
[1] processAction(); /* the controlling method for coordinate input */

Figure 4.21: Class CoordinateInput

```
        for (EVER)
        {
         /* get input action */
         inAction();
         switch(curAct.Type)
         {
           case MOVE: /* set entry coordinates of cursor */
                   switch(curAct.Value)
                   {
                     case LEFT:  curEnt.X -= curDim.Width; break;
                          case RIGHT: curEnt.X += curDim.Width; break;
                          case UP:    curEnt.Y -= curDim.Height; break;
                          case DOWN:  curEnt.Y += curDim.Height; break;
                    }
                        /* redisplay cursor by calling moveCursor method */
                        cursorId-->moveCursor(curEnt,outId);
                        break;

            case ENTER: return(curEnt);

            default: /* ERROR */
                   /* Add error message to display buffer */
                   errBuf-->clearDisplay();
                     errBuf-->text(errMsg);
                     /* Display errBuf */
                   outDev-->displayView(errView);
          }
        }
```

The specification of the StringInput class is exactly the same as the CoordinateInput class, except that it includes the following instance variables:

[a] String inStr; /* the input string */

[b] Integer curPos; /* the current character position in inStr */

[c] ObjId strBuf; /* the display buffer wherein inStr is displayed */

[d] ObjId strView; /* the viewport associated with strBuf */

and its processAction method is described as follows.

```
for (EVER)
{
  /* get input action */
  inAction();
  switch(curAct.Type)
  {
     case MOVE: /* set entry coordinates of character cursor */
                switch(curAct.Value)
                {
                   case LEFT:  curEnt.X -= curDim.Width; curPos--; break;
                         case RIGHT: curEnt.X += curDim.Width; curPos++; break;
                      }
                       /* redisplay cursor by calling moveCursor method */
                       cursorId-->moveCursor(curEnt,outId);
                       break;

      case ADD: /* add character to string */
                 inStr-->addCharacter(curAct.Value);
                 curPos++;
                       /* echo print character */
                 strBuf-->text(curAct.Value);
                       outDev-->displayView(strView);
                       break;
      case DELETE: /* delete current character */
                   inStr-->deleteChar(curPos);
                         /* redisplay string */
                   strBuf-->clearDisplay();
                   strBuf-->text(inStr);
                         outDev-->displayView(strView);
                         break;
      case ENTER: return(inStr);
      default: /* ERROR */
               /* Add error message to display buffer */
               errBuf-->clearDisplay();
                  errBuf-->text(errMsg);
                  /* Display errBuf */
               outDev-->displayView(errView);
    }
 }
```

The specification of class FuncKeyInput is also similar to class CoordinateInput, except that it does not require any cursor related instance variables and that its processAction method is redefined as follows.

```
for (EVER)
{
 /* get input action */
 inAction();
 if (curAct.Type == ERROR)
 {
   /* Add error message to display buffer */
    errBuf-->clearDisplay();
    errBuf-->text(errMsg);
    /* Display errBuf */
    outDev-->displayView(errView);
  }
  else
   return(curAct.Value);
}
```

4.4.4. Mapper Objects

Mapper objects transform user input returned by InputTech objects to internal values operable on by the interactive system. The design of the parameter specification aspect of user interfaces requires three types of mappers, **SelectMapper, PositionMapper,** and **TextMapper,** for each of the interaction tasks supported by the MIMD UIDE. Each mapper provides the facilities to map all possible user inputs returned by InputTech objects to values known to the interaction task objects. For example, SelectMapper accepts any of the three possible user inputs (coordinate, string and function key) and maps them to the corresponding selection, returning the choice to the interaction task object in control. Figure 3.13 (presented in the previous chapter) depicts the role of the three mappers in parameter specification.

The class Mapper which is the root of the mapper inheritance hierarchy, specifies the properties common to all mappers. The following instance variables are common to all mappers:

[a] ObjId viewId; /* the viewport wherein user input is made */

[b] ObjId structId; /* the interactive data structure associated with viewId */

Class SelectMapper is derived from class Mapper and defines the following methods to convert user input into an internal entity. In the following, it is assumed that structId contains user defined name mapping objects which define at least two fields, IntName and ExtName, the internal and external names of selections, respectively.

[1] xyToSel(xyPair); /* map xyPair to selection */

described as follows:

```
/* map xyPair from device coordinates to world coordinates */
intXy = viewId-->viewToWin(xyPair);
/* identify node in structId */
nodeId = structId-->searchDisplay(intXy);
/* access internal name of nodeId and return it */
return((nodeId-->accessInternal())-->retrieveIntName());
```

[2] strToSel(aString); /* map aString to selection */

defined as follows:

```
/* search for node in structId with IntName equal to aString */
codeBlock = [eachNode do:
        if ((eachNode-->accessInternal())-->retrieveExtName == aString)
          return(eachNode);
      ]
/* call iterator method of structId */
nodeId = structId-->eachNodeDo(codeBlock);
/* access internal name of nodeId and return it */
return((nodeId-->accessInternal())-->retrieveIntName());
```

[3] lblToSel(lblId); /* map label identifier to selection */

described similar to the strToSel method.

Class PositionMapper contains the definition of method extToInt(xyPair) which accepts an (x,y) coordinate in device coordinates, calls viewId's viewToWin method to transform xyPair to world coordinates, and returns these transformed coordinates.

The specification of the TextMapper class is depicted in Figure 4.22. The main aim of this class is to tokenize a string (usually a command) input by the user into internal entities. The string is broken down into components corresponding to internally known entities (or keywords) such as function name, service name, language entity name, etc. The tokenized string is maintained in variable keyTknTbl as (keyword,value) pairs. The lexAnalyzer method performs the lexical analysis, returning the next token within inStr. This method uses lexStruct to determine the

Class: TextMapper
SuperClass: Mapper

Instance Variables:

```
[a] String inStr; /* the input string to be tokenized */
[b] ObjId tokenSet; /* the structure that defines associations between
                    keywords and tokens */
[c] ObjId lexStruct; /* the structure used by the lexical analyzer to
                     tokenize inStr */
[d] ObjId keyTknTbl; /* the structure which holds the result of tokenizing
                     inStr as (keyword,value) pairs */
[e] Token curTkn; /* the current token defined as (Type,Value) pair */
[f] Integer curPos; /* the current position within inStr */
```

Methods:

```
[1] lexAnalyzer(); /* the lexical analyzer */

[2] tokenizeStr(aString); /* the controlling method that tokenizes aString */
```

Figure 4.22: Class TextMapper

next token which is returned in curTkn. The method tokenizeStr is described as follows.

```
inStr = aString;
curTkn = lexAnalyzer();
while (curTkn.Type != END)
{
 /* search for associated keyword for token type */
 keyWord = tokenSet-->searchNode(curTkn.Type);
 /* add keyword and token value to keyTknTbl */
 keyTknTbl-->addNode(keyWord,curTkn.Value);
}
return(keyTknTbl);
```

4.4.5. Interaction Task Objects

The preceding sections have developed the classes that form the basic building blocks necessary to define interaction task objects. As depicted in Figure 3.13 (presented in the previous chapter), interaction task objects control an InputTech object and a corresponding Mapper object. The InputTech object is in charge of getting user input. The user input is then passed to the Mapper object which transforms it into an appropriate internal value. This value is returned to the service interface object to process.

The design of interaction tasks defines class InteractTask as the root of the inheritance hierarchy which describes the properties of interaction tasks. The instance variables defined by class InteractTask are the identifiers of the associated InputTech and Mapper objects, inTechId and mapId respectively. The method userInput is used by all subclasses to get the user's input and is defined as

```
userInp = inTechId-->processAction().
```

The userInp variable contains the user input. Class InteractTask also defines a method processInput (as subClassResponsibility) to force its subclasses to define an appropriate method to process the user's input.

The specification of classes SelectTask, PositionTask and TextTask redefine the processInput method to describe the manner in which the class processes the user's input. The SelectTask class defines an instance variable selType which identifies the type of selection; that is, one of COORDINATE, STRING and FUNC_LABEL. Its processInput method is described as follows.

```
/* get user input */
userInput();
/* call appropriate mapping function */
switch (selType)
{
 case COORDINATE: return(mapId-->xyToSel(userInp));
 case STRING:     return(mapId-->strToSel(userInp));
 case FUNC_LABEL: return(mapId-->lblToSel(userInp));
}
```

The PositionTask class defines its processInput method as follows.

```
/* get user input */
userInput();
/* call mapping function */
return(mapId-->extToInt(userInp));
```

Finally, class TextTask's processInput method is defined as

```
/* get user input */
userInput();
/* call mapping function */
return(mapId-->tokenizeStr(userInp));
```

4.4.6. Service Interface Objects

The Service class defines the properties of objects that interface with a particular service provided by the associated function. The main objective of service interface objects is to allow users to specify parameters through interaction task objects, call the appropriate service, and designate the next action to be performed. Figure 4.23 depicts the specification of class Service. Its serve method is appropriately described as follows.

Class: Service
SuperClass: Object

Instance Variables:
[a] String serviceName; /* the internal name of the service */
[b] Function seviceId; /* the actual internal function providing the service */
[c] Object parmStr; /* the actual parameter passed to serviceId */
[d] ObjId errBuf; /* the error buffer passed to serviceId */
[e] ObjId infoBuf; /* the result buffer passed to serviceId */
[f] ObjId errView; /* the viewport associated with errBuf */
[g] ObjId infoView; /* the viewport associated with infoBuf */
[h] Action errAct; /* the action to be taken on error */
[i] Action nextAct; /* the next action to be performed */
[j] ObjId taskId; /* the associated interaction task */

Methods:
[1] serve(); /* the controlling method that provides the service */

Figure 4.23: Class Service

```
/* get parameter specified by user */
parmStr = taskId-->processInput();
/* call actual service */
isError = serviceId(parmStr,errorBuf,infoBuf);
/* process error */
if (isError == TRUE)
{
  /* display errorBuf */
  errView-->displayImage();
  /* return error action */
  return(errAct);
}
else
{
  /* display infoBuf */
  infoView-->displayImage();
  /* return next action */
  return(nextAct);
}
```

Note that both errAct and nextAct designate the next action to be performed, which is either the identifier of a service provided by the function or quitting the function.

4.4.7. Function Interaction Objects

Function interaction objects control the services provided by functions of the interactive system. The specification of class Function is depicted in Figure 4.24. The serviceTbl lists the

Class: Function
SuperClass: Object

Instance Variables:
[a] String funcName; /* the internal name of the function */
[b] Constant quitCmd; /* the action to quit the function */
[c] List serviceTbl; /* a list of services provided by the function */
[d] ObjId curService; /* the current service object in control */

Methods:
[1] doService(); /* controls services of the function */

Figure 4.24: Class Function

identifiers of the service interface objects associated with the function. The main method doService is described as follows.

```
/* get first service */
curService = serviceTbl-->accessNode(1);
for(EVER)
{
  /* call service's serve method */
  nextAct = curService-->serve();
  /* if next action is QUIT, return */
  if (nextAct == quitCmd)
     return;
  else
    curService = serviceTbl-->accessNode(nextAct);
}
```

The overall inheritance hierarchy of the classes that define display objects and function interaction objects is depicted in Figure 4.25.

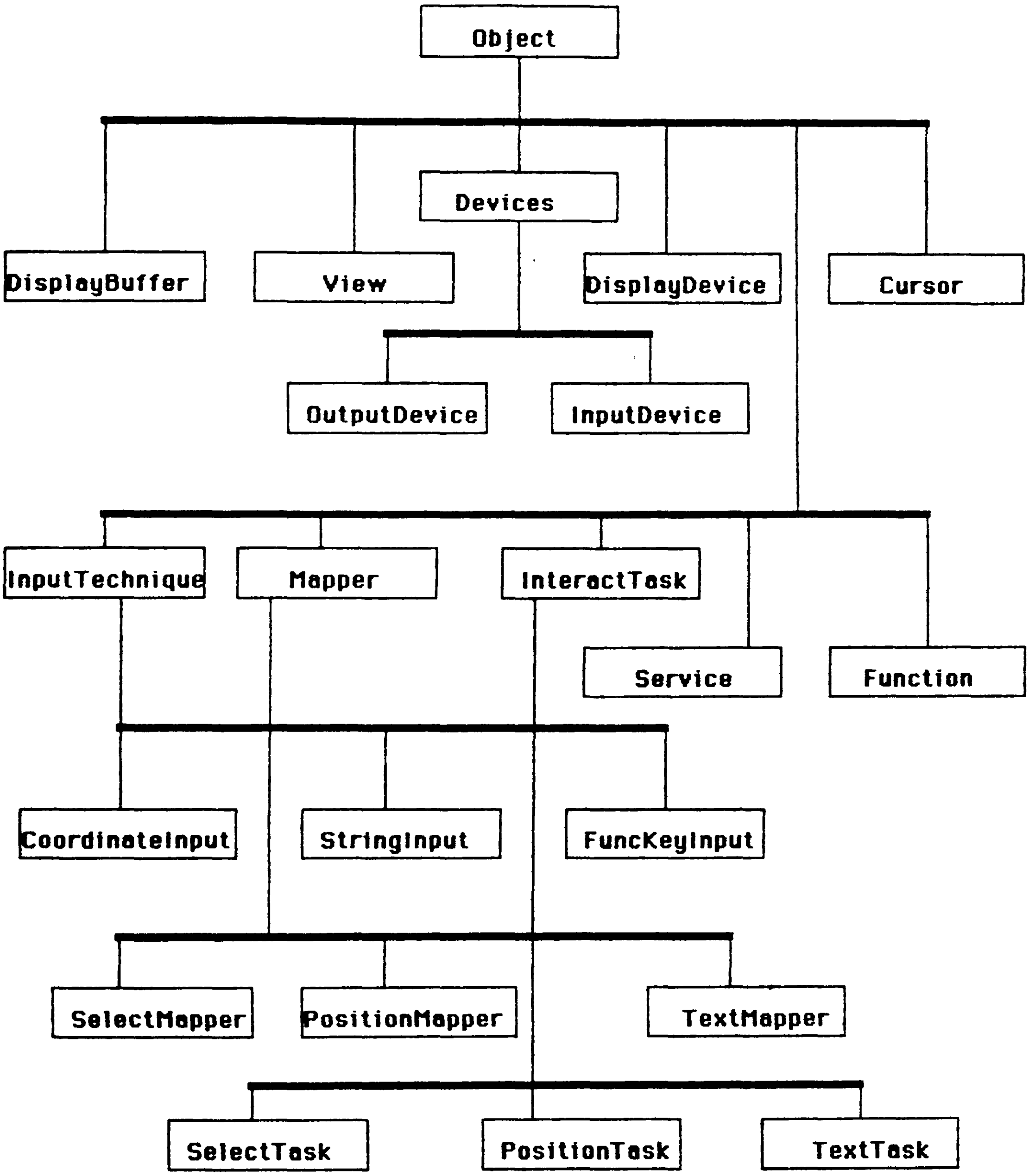


Figure 4.25: User Interface Class Hierarchy

CHAPTER 5

IMPLEMENTING A PROTOTYPE MIMD USER INTERFACE DEVELOPMENT ENVIRONMENT

This chapter describes our experiences in implementing a prototype of the MIMD UIDE developed in the previous chapters. Initially, the major concern of this implementation phase was the quest for an ideal development environment. We realized that implementing an MIMD UIDE would be extremely difficult, if not impossible, without the right set of development tools.

Our requirements for the development environment were straightforward: an object-oriented environment with an appropriate base graphics and windowing system (hereafter referred to as the windowing system). Specifically, the object-oriented environment should provide the necessary support to develop the classes presented in the previous chapters. The windowing system needed for a MIMD UIDE should provide basic facilities for graphics input/output as well as tools to easily build higher level interface objects, such as menus. The development environment should not impose its own philosophy on data structures and the windowing kernel.

Object-oriented development environments can be divided into two broad categories: true object-oriented environments and environments that extend conventional languages (such as Ada, Pascal, Lisp and C) with object-oriented features (e.g., classes, inheritance, etc.). We evaluated a typical environment from each of the categories in an attempt to determine the feasibility of using them to implement the MIMD UIDE prototype. Smalltalk [79, 87], currently the most popular true object-oriented environment, is highly interactive and includes state-of-the-art tools for object-oriented software development. C++ [94] is built as a preprocessor for the popular C programming language [92], and provides conventional programming tools (editor, compiler, linker, debugger) for software development. Smalltalk and C++ represent the two extremes among the existing environments for object-oriented software development.

The next section contrasts these two environments with the aim of showing their strengths and weaknesses as suitable environments for developing the MIMD UIDE. This section substantiates our implementation plans, that is, attempting to modify Smalltalk's SIMD user interface philosophy to achieve a MIMD UIDE.

The rest of the sections within this chapter describe experiences with modifying the very heart of the Smalltalk environment. The second section concerns implementing interactive data structures within Smalltalk. The following section is devoted to our attempts in modifying Smalltalk's Model-View-Controller user interface framework towards an MIMD framework.

5.1. Contrasting Smalltalk vs. C++

In the previous chapter, the object-oriented philosophy was introduced in terms of the basic components: objects, classes, methods and message passing. Inheritance was also introduced as the basis for object-oriented software development. It is important to note that every object-oriented development environment must provide the above basic components as well as inheritance.

There exist, however, other features and techniques that determine the effectiveness of the support provided by an environment for object-oriented software development. Johnson and Foote [95] identify the following features and techniques as crucial to effective object-oriented software development: **polymorphism, standard protocols, frameworks** and **tool-kits.** Each of these features has a profound effect on the reusability and extendibility of the object-oriented system. The following discusses each of these in detail.

Conventional typed languages, such as Pascal and C, are **monomorphic** in the sense that every object (function, variable, constant, etc.) has a unique type. Statically typed languages impose a further restriction by requiring that the types of all objects be known at compile time. Such a strong restriction is a double-edged sword for software development. On one hand it allows type inconsistencies to be discovered at compile time and can guarantee that executed programs are type consistent. Furthermore, strong static typing facilitates efficient code generation and also imposes a strong discipline on programming. Conversely, static typing decreases flexibility and expressive power by prematurely constraining the behavior of objects. In contrast, **polymorphism** increases flexibility and expressive power by deferring the determination of the type (behavior) of objects to execution time. Thus, a polymorphic operation can be viewed as having multiple behaviors depending on the type of its operands [96]. A popular example of a polymorphic operation is sorting which should be applicable to a wide range of types. Thus, polymorphism facilitates the development of generalized abstract operations, which in turn produce software products that promote extensibility and reusability.

Through the years many techniques have been developed to allow limited polymorphism in conventional languages. **Overloading** function names allows the same function name to be developed for a large number of types. For example, the operation **a + b** would invoke an appropriate + function depending on the type of the operands. Furthermore, type coercion allows operations to be more polymorphic, that is, applicable to a variety of compatible types. **Generics** [97] facilitate the definition of a template of a general operation which is not bound to any type. Specific instances of the general operation can be instantiated by providing type information.

Polymorphism is the cornerstone for developing effective object-oriented software. Note that in an object-oriented system, operations are performed by message passing. Messages in turn determine the correct method (operation) in the receiver's class and invokes that method. This implies that messages are dynamically bound procedure calls. Thus message sending causes polymorphism. Another perspective of object-oriented polymorphism is that the type of an object is determined by the set of messages it understands, that is, its behavior.

This general object-oriented polymorphism is an integral part of the development of interactive systems. Data structures depend on polymorphism to maintain and manipulate heterogeneous objects. Windowing systems in general depend on polymorphism to allow different types of objects to be displayed. The only requirement in both cases is that the objects understand some common set of messages.

The underlying basis of abstract types and hence object-oriented systems is the separation of the specification from the implementation of the object. The specification of the object defines its behavior or **protocol.** Developing generalized abstract behavior requires **standard protocols.** Objects with standard (or identical) protocols are interchangeable or **plug compatible.** That is, objects can be substituted or changed only with objects that have an identical protocol. Note that this is more general that allowing substitutions only between objects derived from similar classes. Standard protocols allow complex classes to be built by interconnecting a set of plug compatible objects. Note that the MIMD UIDE is built as complex classes that interconnect compatible objects. Furthermore, standard protocols promote a standard vocabulary for communication between object-oriented programmers. This is extremely important in the realm of object-oriented programming, since it facilitates easy learning and reuse of classes.

Abstract classes provide an important vehicle for representing standard behaviors or protocols [95, 98]. Abstract classes were used extensively in the design of the MIMD UIDE presented in the previous chapter. The major characteristics of abstract classes are as follows. Abstract classes never define data (or instance variables) but the standard behavior (or methods) of classes that are derived from it. Note that abstract classes never have instances. However, **concrete** classes derived from it have instances (or objects). Thus, the roots of class hierarchies should always be abstract, while the leaf classes are always concrete. It is important to note that inheriting from an abstract class can never violate encapsulation. A good example of an abstract behavior is enumeration for data structures. A number of standard protocols can be defined for data structures dependent on an iterator. For example, Figure 5.1 shows standard protocols for enumeration provided by Smalltalk's Collection class that depend on a **do:** (iterator) method that is defined at the concrete subclass level. As an example, the iterator method

```
enumerating
   do: aBlock
      self subclassResponsibility
   collect: aBlock
      | newCollection |
      newCollection ← self species new.
      self do: [ :each | newCollection add: (aBlock value: each)].
      ↑newCollection
   detect: aBlock
      ↑self detect: aBlock ifNone: [self errorNotFound]

  detect: aBlock ifNone: exceptionBlock
     self do: [ :each | (aBlock value: each) ifTrue: [↑each]].
     ↑exceptionBlock value
  inject: thisValue into: binaryBlock
     | nextValue |
     nextValue ← thisValue.
     self do: [ :each | nextValue ← binaryBlock value: nextValue value: each].
     ↑nextValue
  reject: aBlock
     ↑self select: [ :element | (aBlock value: element) == false]
  select: aBlock
     | newCollection |
     newCollection ← self species new.
     self do: [ :each | (aBlock value: each) ifTrue: [newCollection add: each]].
     ↑newCollection
```

Figure 5.1: Collection's Enumeration Protocol

for the concrete class Set is shown below:

```
do: aBlock

1 to: self basicSize do:
   [:index |
         (self basicAt: index) isNil
            ifFalse: [aBlock value: (self basicAt: index)]]
```

Note that the abstract classes correspond closely to the conventional notion of program skeletons which require certain parts to be defined to generate the concrete program.

The discussion so far has concentrated on features and techniques for developing general abstract classes and their immediate descendents. However, effective software development is mainly concerned with developing general abstract designs. Thus, the design of software is usually defined in terms of the major components and how they interact. The abstract design of an object-oriented software system is called a **framework.** Ideally, a framework is a set of abstract classes, one for each major component of the design. A set of messages defines the interfaces between the components. Two examples of frameworks, Macintosh's McApp and Smalltalk's MVC, were presented in Chapter 2 as examples of SIMD UIDEs. Note that frameworks facilitate reuse at the highest level of granularity and are therefore desirable.

The effectiveness of a framework is dependent on the ease with which it can be (re)used. Frameworks can therefore be classified as **white-box** or **black-box** [95]. A white-box framework necessitates knowledge about how it is constructed (that is, its implementation), to be (re)used. A white-box framework provides the top level control and sequencing of activities and can therefore be viewed as an extendible skeleton. Using white-box frameworks requires the creation of many additional (sub)classes to tailor the framework for the particular application. In contrast, a black-box framework only requires the provision of components that define the behavior specific to the application. The black-box framework uses these application specific components to automatically generate the object-oriented software. Thus, black-box frameworks require understanding of only the external interface of the major components. It is important to note that there are only a few examples of frameworks currently available. Most of these available frameworks are white-box and, as we shall show later, very difficult to use.

Frameworks provide the basis for developing **toolkits**, which provide users with a collection of high-level tools to automatically configure and construct new applications. The main advantage of black-box frameworks is that they provide an ideal foundation for developing toolkits. Note that the MIMD UIDE is a toolkit that allows end users to configure and construct user interfaces for interactive applications. It is important to point out that currently there exist only a few instances of object-oriented toolkits. Two that have been discussed in the literature are Glazier [99] and ARK [100]. Both of these are built on top of white-box frameworks.

5.1.1. The Smalltalk Object-Oriented Development Environment

Smalltalk is the most popular true object-oriented environment. A true object-oriented environment implies an everything-is-an-object philosophy, message sending as late bound procedure calls, and general polymorphism as described in the introduction to this section. Thus, an expression such as **a + b** is a message + with argument **b** that is sent to object **a.** Even simple arithmetic such as "3 + 4" is implemented as above.

Smalltalk provides the standard programming syntax available in most conventional programming languages, including assignments, conditionals and loops. However, everything in Smalltalk is implemented in an object-oriented manner. A particularly useful construct is the **block** object, which represents a deferred sequence of actions. For example, the block below is used within a loop to initialize the contents of an array.

```
[index <- index + 1.
 array at: index put: 0].
```

A block is not executed immediately. The block must be requested to execute itself by passing it the message **value.** Furthermore, blocks can have a maximum of two arguments associated with it and can be passed actual arguments during execution by using the **value:** or **value:value:** messages. For example the following code defines a block that determines the size of any array, and is then used on a specific array instance.

```
sizeAdder <-- [:array | total <-- total + array size].
total <-- 0.
sizeAdder value: #(a b c)
```

where #(a b c) creates an array with values a, b, and c.

Loops are implemented as messages in separate classes. For example, a simple Pascal-like **for** loop is implemented in the Integer class by the method **timesRepeat.** Thus, the expression

```
4 timesRepeat: [amount <-- amount + amount]
```

invokes the method timesRepeat in object 4's class passing it the block **[amount <-- amount + amount]** as an argument. The method **timesRepeat: aBlock** is implemented as

```
| count |

count <-- 1.
[count <= self]
   whileTrue: [aBlock value.
               count <-- count + 1].
```

Note that the **whileTrue:** message is part of the protocol for blocks.

The above discussion on blocks serves two purposes. One is to show the complete object-orientedness of the Smalltalk system. The other is to show the power of blocks as objects. Note that the MIMD UIDE design presented in the previous Chapter made extensive use of such a block concept.

The standard Smalltalk environment consists of a large class hierarchy which includes classes that define its data structures and user interfaces. For example, Smalltalk-80 version 2.0 has more than 250 classes with over 2000 methods. To use Smalltalk to develop object-oriented software requires a good understanding of these classes, since a large percentage of development time is spent in reusing or modifying existing classes.

As pointed out in Chapter 2, the Smalltalk environment imposes a consistent interaction style that is depicted in Figure 5.2. Smalltalk is window oriented, presenting many simultaneous contexts. A three-button mouse is used exclusively for selection, usually from pop-up menus that depict the choices for selection. Keyboard input is used only to get names (for example for filing purposes), and for entering text within editors. Note that text editor functions, such as selecting text for replacement within an editor, is also done using the mouse.

Smalltalk is equipped with state-of-the-art tools for object-oriented software development. Browsers allow navigation through the class hierarchy. Inspectors permit the state of instances to be inspected. Editors allow pictures and classes to be edited. Finally, debuggers allow manipulation of the runtime state. The system also provides functions that aid reusability. The **senders** function when applied to a method collects all methods in the system that send that method as depicted in Figure 5.3. This allows the implementor to determine how a particular method is used within the system. The **implementors** function provides a list of all classes that implement similarly named methods, as shown in Figure 5.4. This allows the implementor to determine how standard protocols are implemented within the system.

Thus, Smalltalk is definitely the state-of-the-art in object-oriented development environments. The major drawback, however, in using Smalltalk to develop the MIMD UIDE is its enforcement of a consistent interaction style. Note that the MIMD UIDE requires a redesign of the entire foundation of Smalltalk including data structures, the graphics kernel, the windowing environment and the interaction style. Since Smalltalk's entire class hierarchy is based on its implementation of these base components, reusability is minimized. In fact, building an MIMD UIDE using a SIMD UIDE is futile since we are using an enforced consistent style to break the system to a more generalized style. Such an attempt is possible only if the SIMD UIDE is built on top of a very general abstract foundation. As we shall show later in this chapter, this is not the case in Smalltalk.

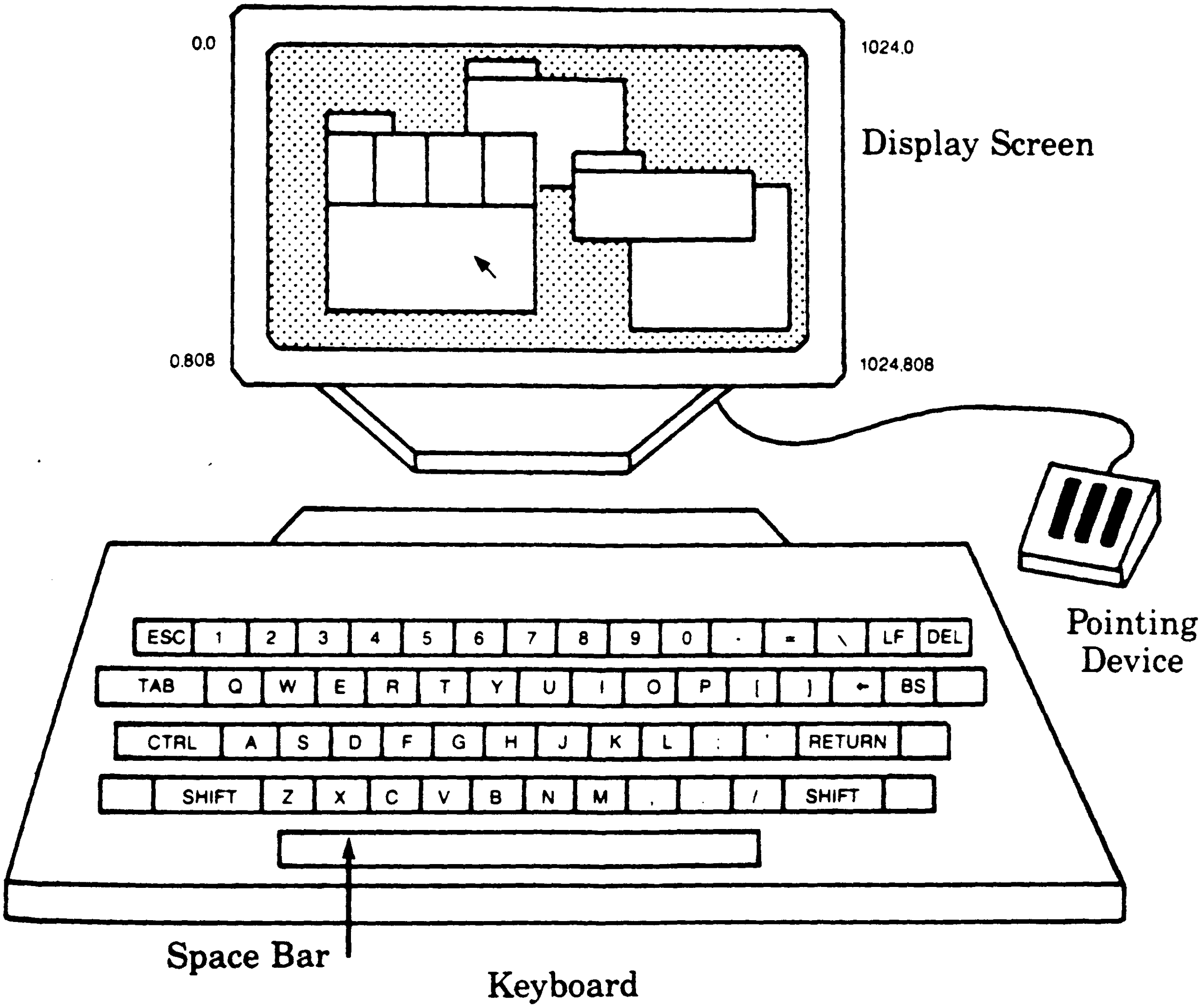
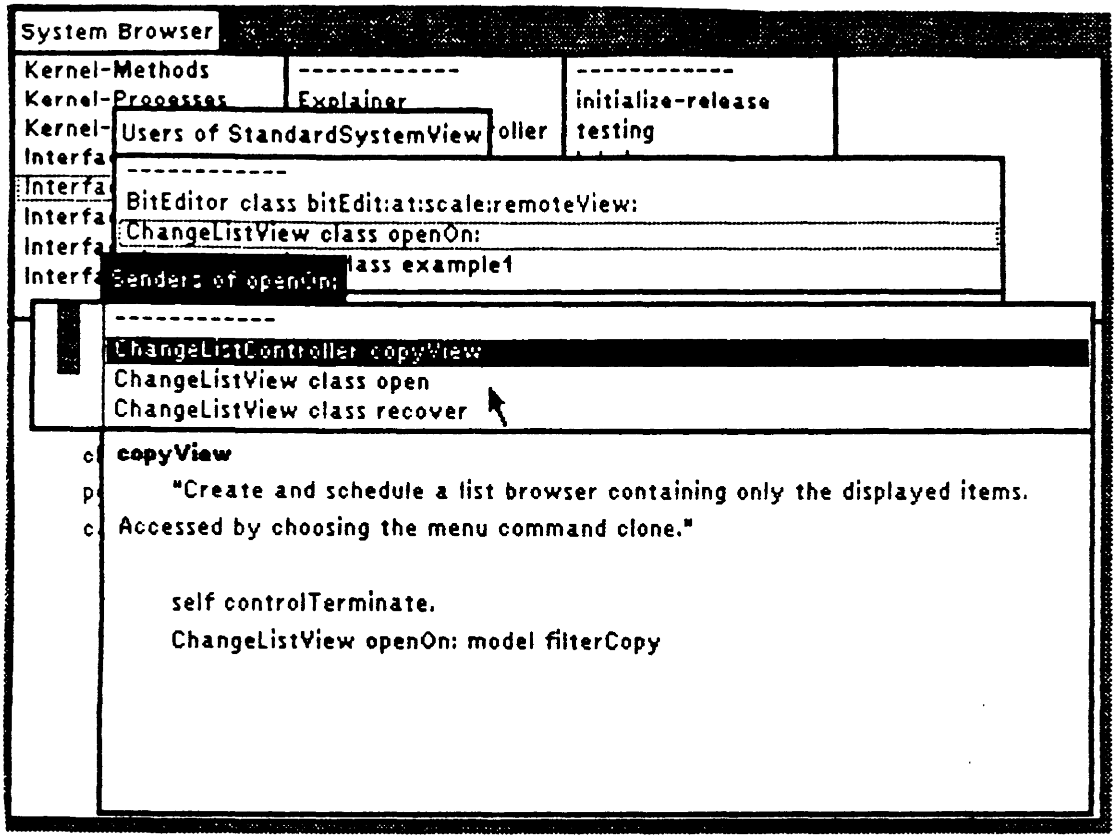
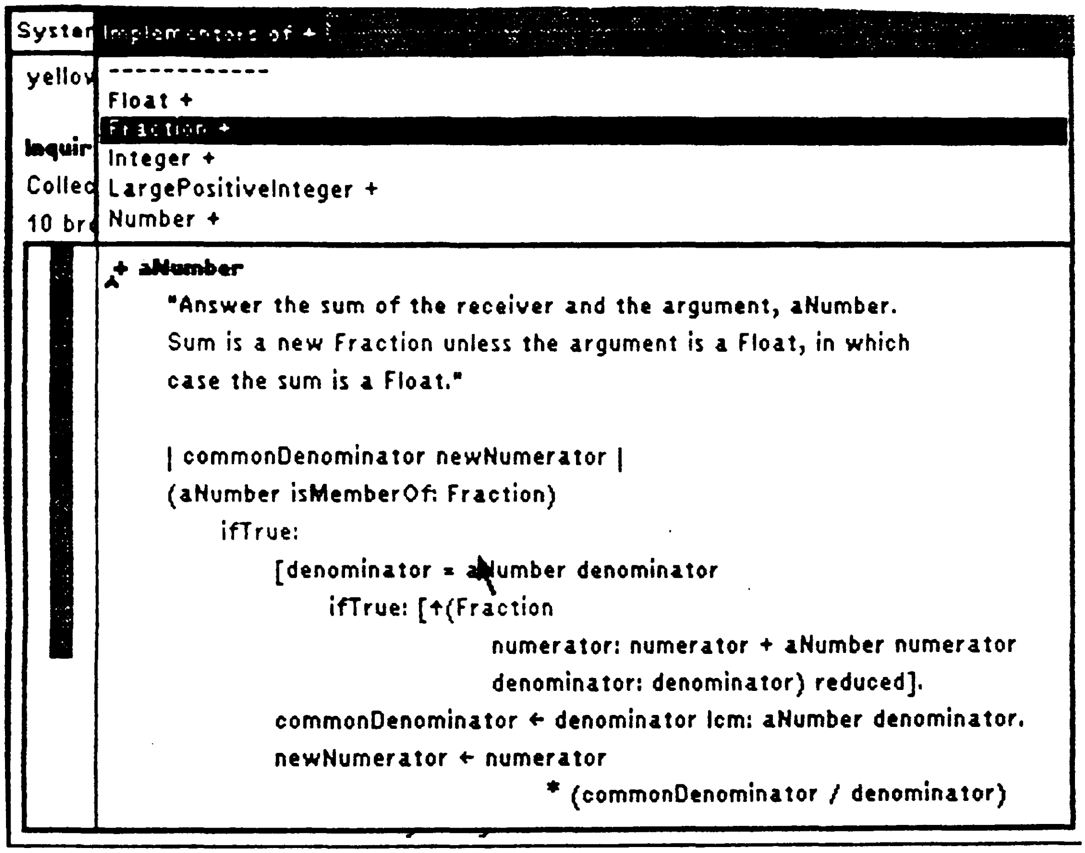

0,0
1024,0
Display Screen
0,808
1024,808
Pointing Device
ESC 1 2 3 4 5 6 7 8 9 0 - = \ LF DEL
TAB Q W E R T Y U I O P [ ] ← BS
CTRL A S D F G H J K L ; ' RETURN
SHIFT Z X C V B N M , . / SHIFT
Space Bar
Keyboard

Figure 5.2: Smalltalk's User Interaction Style



```
copyView
	"Create and schedule a list browser containing only the displayed items.
Accessed by choosing the menu command clone."

	self controlTerminate.
	ChangeListView openOn: model filterCopy
```

Figure 5.3: Example of Sender's Function



```
+ aNumber
    "Answer the sum of the receiver and the argument, aNumber.
    Sum is a new Fraction unless the argument is a Float, in which
    case the sum is a Float."

    | commonDenominator newNumerator |
    (aNumber isMemberOf: Fraction)
        ifTrue:
            [denominator = aNumber denominator
                ifTrue: [↑(Fraction
                            numerator: numerator + aNumber numerator
                            denominator: denominator) reduced].
            commonDenominator ← denominator lcm: aNumber denominator.
            newNumerator ← numerator
                                * (commonDenominator / denominator)
```

Figure 5.4: Example of Implementors Function

5.1.2. The C++ Object-Oriented Language

C++ [94] extends the powerful software development programming language C [92] with object-oriented features. Bjarne Stroustrop [94] aptly describes the power of both C and C++:

> A programming language serves two related purposes: it provides a vehicle for the programmer to specify actions to be executed and a set of concepts for the programmer to use when thinking about what can be done. The first aspect ideally requires a language that is "close to the machine", so that all important aspects of a machine are handled simply and efficiently in a way that is reasonably obvious to the programmer. The C language was primarily designed with this in mind. The second aspect ideally requires a language that is "close to the problem to be solved" so that the concepts of a solution can be expressed directly and concisely. The facilities added to C to create C++ were primarily designed with this in mind.

In the last decade, C has become the language of choice for software development due to its simplicity, its closeness to the machine, its portability, and most importantly, its ability to directly access the powerful UNIX operating system [101]. Consequently, C++ has access to thousands of library functions and utility software code that have been written in C.

Most C++ implementations are built as preprocessors that translate the object-oriented extensions into C text. The object-oriented features make C++ a superset of C, allowing C++ to be similar to C syntactically, and maintains its simplicity and power. The major disadvantages of C++ are the limited support for polymorphism and the conventional environment for developing object-oriented software.

The ANSI standard C (on which C++ is built) is statically typed. This implies that sending a message to an object requires knowledge about the type of the function that implements the method, as well as the types of all the parameters. C++ supports polymorphism in two ways. The first is by overloading function names. Figures 5.5 and 5.6 show the specification and implementation respectively, of class Complex. Arithmetic on complex numbers is achieved by operator overloading. Another feature that supports both standard protocols and polymorphism is **virtual** functions. A function defined as virtual in a class allows subclasses derived from it to provide their own version. The system guarantees that the correct function is applied according to the type of the object. As an example, an abstract class DisplayObject could provide a virtual

```
class complex {
   double realPart, imagPart;

public:
   complex(double x = 0.0, double y = 0.0) /* constructor */
   {   realPart  = x;
       imagPart = y;
   }

   double real()   {return realPart;}
   double imag() {return imagPart;}

   complex operator + (complex&);  /* overload */
   complex operator *  (complex&);  /* operators */
   complex operator + (double&);     /* if double is added */
}
```

Figure 5.5: C++ Specification of Class Complex

```
complex complex::operator + (complex& aComplex)
{   double newReal, newImag;

    newReal = realPart + aComplex.real();
    newImag = imagPart + aComplex.imag();
    return complex(newReal, newImag);
}

complex complex::operator * (complex& aComplex)
{   double newReal, newImag;

    newReal = (realPart * aComplex.real()) -
                 (imagPart * aComplex.imag());
    newImag = (realPart * aComplex.imag()) +
                 (imagPart * aComplex.real());

    return complex(newReal, newImag);
}

complex complex::operator + (double& aNumber)
{   double newReal;

    newReal = realPart + aNumber;

    return complex(newReal, imagPart);
}
```

Figure 5.6: C++ Implementation of Class Complex

function **draw,** indicating that all displayable objects must be able to draw themselves. Any class derived from DisplayObject is therefore forced to provide (define) their draw function.

Note, however, that the above features do not support polymorphism between objects in different class hierarchies. This type of polymorphism is essential in developing heterogeneous data structures and windowing systems. The major problem is that C++ does not support dynamic binding. However, it is possible to simulate dynamic binding in C++ essentially by forcing each class to provide its own **messager.** Thus, invoking a method (function) of a class is achieved by invoking the messager and providing it with the name of the method, an argument list and a return list. The template of the messager is provided below.

```
void messager (char * methodName, void ** argumentList, void ** returnList)

{ int methodIndex = "Search for methodName in Class's list of methods",
  if ("Not Found") then
     return error,
  switch (methodIndex) {
        case METHODONE:
                    "Extract the parameters needed for the method from
                     argumentList and cast it to appropriate type",
                    "Call method with arguments",
                    "Cast returned value into (void *) and attach to
                     returnList",
                     break,
         case METHODTWO: ........
              .
              .
              .
              .
}
```

Note that (void *) is a typeless pointer and is extremely useful in deferring the determination of type to execution time. Its use as a general argument list is similar to the standard C facility of using **argv** and **argc** to accept program arguments for the main function. The messager concept was used successfully in developing a prototype implementation of the data structure of the MIMD UIDE.

Another deficiency in C++ is that it does not support the powerful Smalltalk concept of blocks of statements as objects. As is evident from the design of the MIMD UIDE, such a concept is extremely useful in the design of generalized abstractions. In C++, such blocks were implemented as stand alone functions and passed as parameters.

The only class that is provided with C++ is Stream, which implements type secure, flexible and efficient methods for standard (non-graphical) input and output. This classless property of C++ is extremely attractive for developing an MIMD UIDE since there are no biases imposed by the underlying system. It is also important to note that C++ provides no graphics/windowing support. This is again an advantage as far as developing the MIMD UIDE is concerned because it allows any available graphics/windowing system to be used with the system.

Finally, the development environment of C++ is conventional, that is, an editor, compiler, linker, and symbolic debugger. There are no special tools/functions provided to support object-oriented programming. While this is clearly a negative incentive, it is important to point out that in the recent OOPSLA'88 conference, numerous vendors were exhibiting C++ development environments that were similar to Smalltalk.

5.1.3. Discussion

The crucial aspect of implementing the MIMD UIDE according to the design presented in the previous chapter is the underlying graphics and windowing system. An ideal graphics/windowing system provides generalized components (e.g., input/output devices, points, text, windows, etc.) and most importantly does not impose any style of interaction. It should also provide facilities to build higher level components (e.g., menus, editors, icons, etc.).

Smalltalk is therefore extremely unsuitable as a development environment for the MIMD UIDE. Its base graphics/windowing system imposes a strict adherence to a consistent style, which is reflected throughout the class hierarchy. Moreover, the underlying graphics kernel is specific to this style and does not support generalized components. Thus, creating a MIMD UIDE using a Smalltalk-like SIMD environment requires the replacement or modification of core classes which would invalidate a large percentage of the existing class hierarchy. In fact, as shall be shown later, such massive reorganization is extremely difficult and is not supported by current object-oriented technology.

C++ on the other hand is classless and provides no support for graphics and windowing. It provides the base facilities for object-oriented development and allows any graphics/windowing system to be attached to it. This makes C++ a double-edged sword for the development of the MIMD UIDE. On one hand, it presents an ideal environment since it imposes no biases. On the other hand, building an entire MIMD UIDE from scratch has many disadvantages.

The major disadvantage is in finding an ideal graphics/windowing system. Most of the currently available graphics/windowing systems impose their own biases, and are not general enough to implement a MIMD UIDE according to the design presented in the previous chapter. In fact, many graphics/windowing systems were evaluated including Starbase [102], MetaWindows [103], and Domain Graphics [104]. None of these systems were found to be suitable as a base for the MIMD implementation. In Chapter 2, however, it was noted that X windows [105] is gaining rapid popularity as a standard due to its generality. X may prove to be an ideal choice for the underlying graphics/windowing system for the MIMD UIDE.

The other disadvantage of using C++ is its conventional development environment. Such an environment makes a large object-oriented implementation extremely tedious and time consuming, mainly because of the lack of support for rapid prototyping, reusability and interactive debugging.

We realized that developing the MIMD UIDE in C++ requires a substantial investment in building the base line classes. With our limited experiences in building object-oriented software systems, we felt that much more could be learned by attempting to modify an existing, fully operational UIDE. Furthermore, if the fully operational UIDE employed a SIMD framework then its modification would present an invaluable basis for determining the effectiveness of our MIMD UIDE design.

Smalltalk's Model-View-Controller (MVC) user interface framework is currently the most popular and most effective UIDE. We briefly presented the MVC framework in Chapter 2 as an example of a SIMD UIDE. Attempting a modification of the MVC framework presented a golden opportunity to contrast and evaluate our MIMD UIDE. The major goals of this attempt were:

[a] to understand the complexity of building object-oriented user interface development environments,

[b] to determine the characteristics of an ideal graphics/windowing system suitable for an MIMD UIDE implementation,

[c] to contrast MIMD UIDEs with SIMD UIDEs, and

[d] to determine the strengths and weaknesses of the MIMD UIDE design.

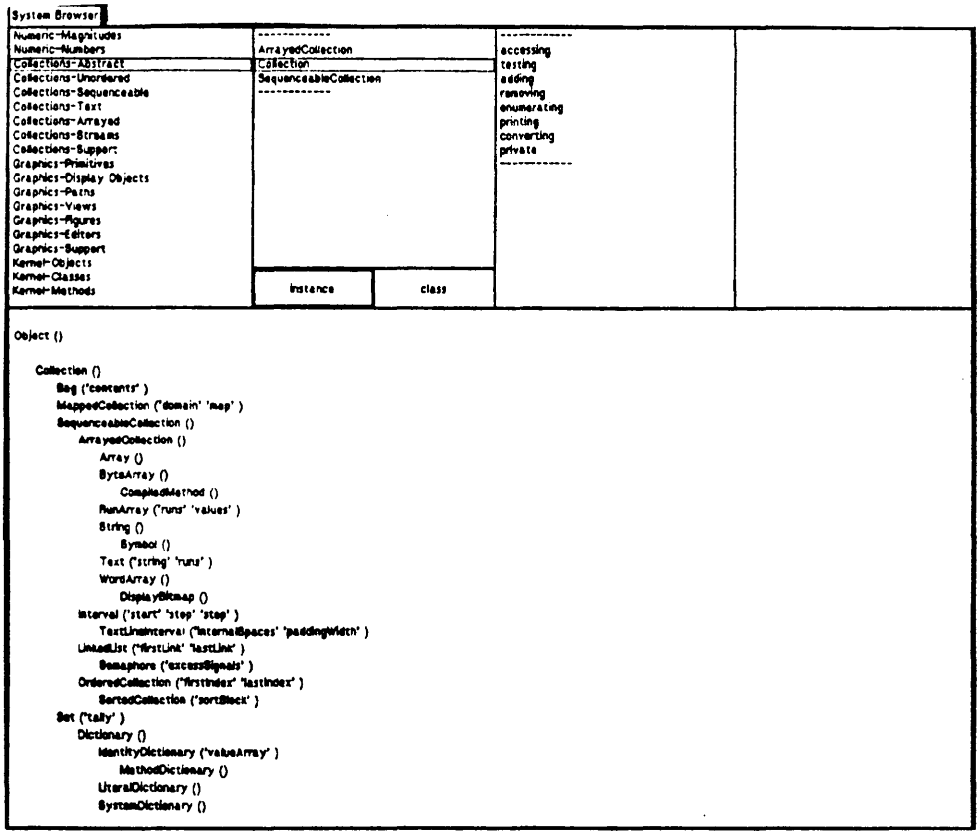
The rest of this chapter discusses the efforts in making Smalltalk an MIMD UIDE.

5.2. The Interactive Data Structures Prototype

Smalltalk's data structure hierarchy is shown in Figure 5.7. Class Collection provides the standard protocol for all data structures, as shown in Figure 5.8. Class Collection is an abstract class, and some of its methods (such as iteration (do:), adding elements (add:) and removing objects (remove:ifAbsent:)) must be provided by the subclasses derived from it.

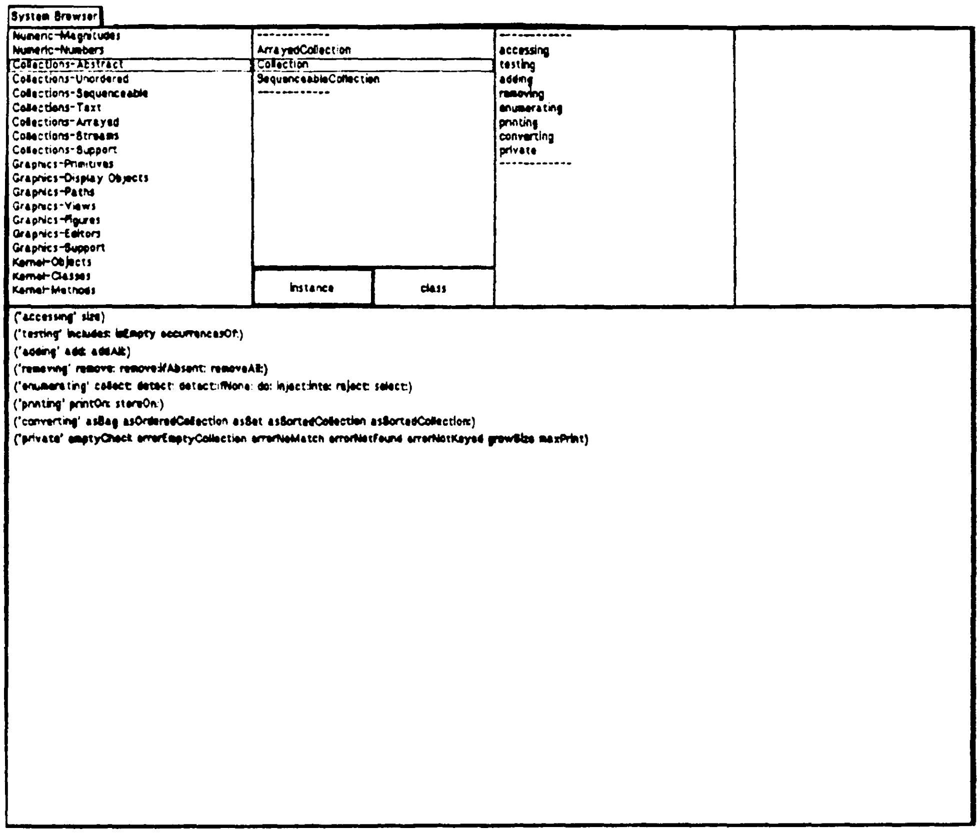
Figure 5.9 shows a map for distinguishing between the various data structures provided by the system. Note that all the data structures provided are essentially lists, and the differences between them depends on the implementation of the data elements or the structure. Smalltalk's data structure hierarchy is not suitable as a basis for interactive data structures within the MIMD UIDE because of the following reasons.

[1] The major advantage of using an object-oriented implementation is the separation of the implementation from the specification. The user of a list structure should not be concerned with the internal representation of a list, e.g., whether an array or linked list. Thus, the Collection class is not general enough to model the well established conventional abstract data structures [106]. Note that the user of Smalltalk's data structure hierarchy must have intimate knowledge of the implementation of the internal representations.



```
Object ()

    Collection ()
        Bag ('contents' )
        MappedCollection ('domain' 'map' )
        SequenceableCollection ()
            ArrayedCollection ()
                Array ()
                ByteArray ()
                    CompiledMethod ()
                RunArray ('runs' 'values' )
                String ()
                    Symbol ()
                Text ('string' 'runs' )
                WordArray ()
                    DisplayBitmap ()
            Interval ('start' 'stop' 'step' )
                TextLineInterval ('internalSpaces' 'paddingWidth' )
            LinkedList ('firstLink' 'lastLink' )
                Semaphore ('excessSignals' )
            OrderedCollection ('firstIndex' 'lastIndex' )
                SortedCollection ('sortBlock' )
        Set ('tally' )
            Dictionary ()
                IdentityDictionary ('valueArray' )
                    MethodDictionary ()
                LiteralDictionary ()
                SystemDictionary ()
```

Figure 5.7: Collection Class Hierarchy



```
('accessing' size)
('testing' includes: isEmpty occurrencesOf:)
('adding' add: addAll:)
('removing' remove: remove:ifAbsent: removeAll:)
('enumerating' collect: detect: detect:ifNone: do: inject:into: reject: select:)
('printing' printOn: storeOn:)
('converting' asBag asOrderedCollection asSet asSortedCollection asSortedCollection:)
('private' emptyCheck errorEmptyCollection errorNoMatch errorNotFound errorNotKeyed growSize maxPrint)
```

Figure 5.8: Collection Class Protocol

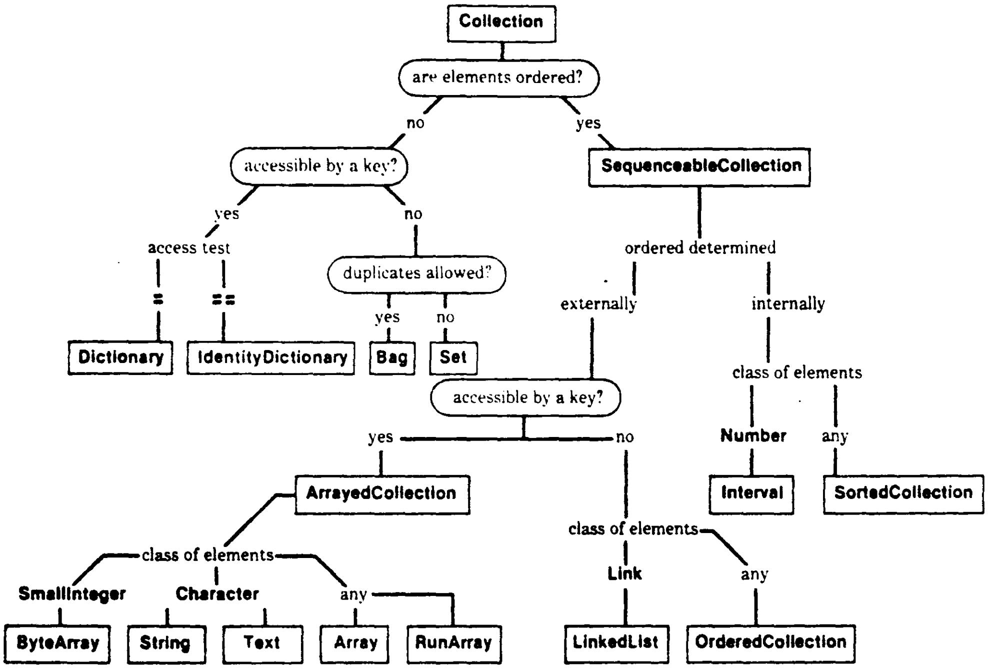


Figure 5.9: Smalltalk's Data Structure Map

[2] The data structure hierarchy provides no support for other types of structures, such as trees and graphs. The class hierarchy does not have support for queues or stacks, even though these are just specialized lists. The incorporation of queues or stacks within the hierarchy requires a new subclass that specializes each of the concrete structure classes. [The abstract Collection classes are SequencableCollection and ArrayedCollection.] This again indicates a malformed general data structure abstraction.

[3] The data structure hierarchy provides no support for external representations. Its main purpose is to model the internal representation of the basic data structures based on lists.

[4] There are other indications that the Collection hierarchy needs reorganization. For instance, consider the size method which is used to return the size of the data structure. This method is implemented in class Collection as follows.

```
| tally |
tally <-- 0.
self do: [:each | tally <-- tally + 1].
^tally
```

This indicates that the default size for each structure is the number of elements it contains. However, all of Collection's immediate subclasses (that is, SequencableCollection, Arrayed-Collection, Bag, MappedCollection and Set) redefine the size method. Since Collection is an abstract class, its default size behavior is never used in any data structure objects. Taking this inconsistency one step further, the abstract class SequencableCollection redefines the size method as **subClassResponsibility** forcing all of its subclasses to provide a size method. Note that SequencableCollection is ensuring that none of its subclasses can access the default behavior defined in Collection. However, SequencableCollection's subclass LinkedList provides a size method that is exactly the same as the default behavior in Collection!! This simple example of inconsistency shows the need for reorganizing Smalltalk's data structure hierarchy.

The interactive data structure prototype was therefore developed parallel to the Collection class. It is important to note that it was necessary to use Smalltalk's data structures in the implementation. The prototype followed the design exactly. Smalltalk's graphics kernel provided the basis for developing the external representations. These classes were used as follows.

[1] The XyPair type used throughout the MIMD design was replaced by Smalltalk's Point class. The protocol for class Point is shown in Figure 5.10 Note that the methods match the requirements for XyPair perfectly.

[2] The Dimension type used throughout the MIMD design was replaced by Smalltalk's Rectangle class. The protocol for class Rectangle is shown in Figure 5.11 Again there was an exact match between Dimension and Rectangle.

[3] The external representation of the interactive data structures was developed using classes from Smalltalk's graphics kernel. Smalltalk images are represented as instances of class Form. An example of a Form is presented in Figure 5.12. A Form has a rectangular structure that defines its width and height, and a bitmap which stores the pattern of pixels that define the contents of the form. Any graphical object, including text, can be displayed on a Form. Forms can be combined into a single larger Form as depicted in Figure 5.13. Since the basic representation of Forms is a bitmap, many sophisticated operations can be performed. Forms can be copied between source and destination Forms with different combination rules (replace, over, under, etc.), can be clipped to a particular rectangular area, and can be filled with patterns defined by a mask Form.

Forms were used to represent the DisplayBuffer class within the MIMD UIDE design. Thus, the external representations of interactive structures are maintained as Forms. Since Forms can be made up of smaller Forms, it is appropriate to represent the display of each ExternalData object as a Form. These ExternalData Forms are then displayed on the DisplayBuffer Form to create the external representation of the interactive structure. Finally, the displayGeometry instance variable was implemented as a Smalltalk block

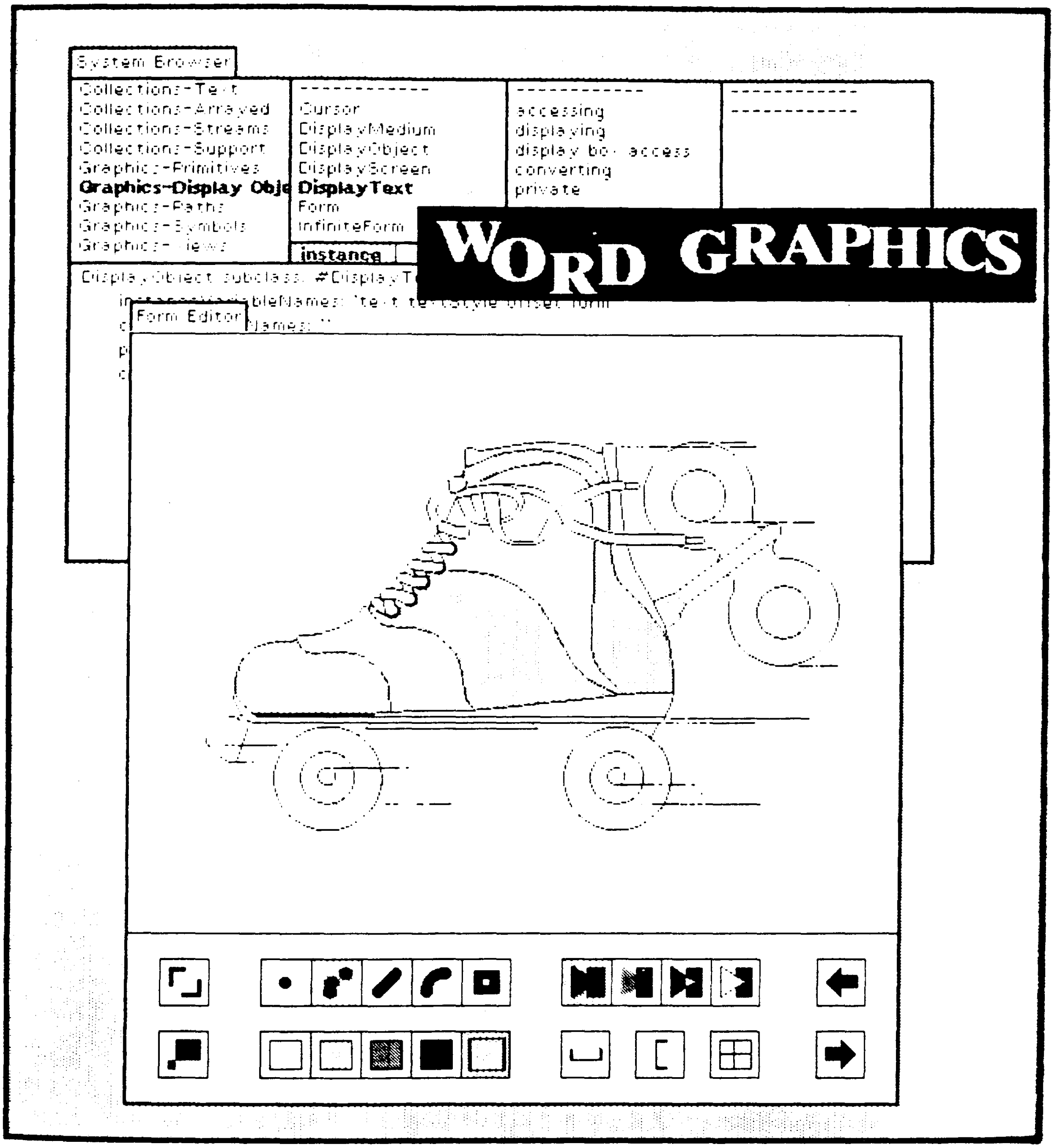
Point instance protocol	
accessing	
x	Answer the x coordinate.
x: aNumber	Set the x coordinate to be the argument, aNumber.
y	Answer the y coordinate.
y: aNumber	Set the y coordinate to be the argument, aNumber.
comparing	
< aPoint	Answer whether the receiver is above and to the left of the argument, aPoint.
< = aPoint	Answer whether the receiver is neither below nor to the right of the argument, aPoint.
> aPoint	Answer whether the receiver is below and to the right of the argument, aPoint.
> = aPoint	Answer whether the receiver is neither above nor to the left of the argument, aPoint.
max: aPoint	Answer the lower right corner of the rectangle uniquely defined by the receiver and the argument, aPoint.
min: aPoint	Answer the upper left corner of the rectangle uniquely defined by the receiver and the argument, aPoint.
point functions	
dist: aPoint	Answer the distance between the argument, aPoint, and the receiver.
dotProduct: aPoint	Answer a Number that is the dot product of the receiver and the argument, aPoint.
grid: aPoint	Answer a Point to the nearest rounded grid modules specified by the argument, aPoint.
normal	Answer a Point representing the unit vector rotated 90 deg clockwise.
transpose	Answer a Point whose x is the receiver's y and whose y is the receiver's x.
truncatedGrid: aPoint	Answer a Point to the nearest truncated grid modules specified by the argument, aPoint.

Figure 5.10: Point Class Protocol

Rectangle instance protocol

accessing	
topLeft	Answer the Point at the top left corner of the receiver.
topCenter	Answer the Point at the center of the receiver's top horizontal line.
topRight	Answer the Point at the top right corner of the receiver.
rightCenter	Answer the Point at the center of the receiver's right vertical line.
bottomRight .	Answer the Point at the bottom right corner of the receiver.
bottomCenter	Answer the Point at the center of the receiver's bottom horizontal line.
bottomLeft	Answer the Point at the bottom left corner of the receiver.
leftCenter	Answer the Point at the center of the receiver's left vertical line.
center	Answer the Point at the center of the receiver.
area	Answer the receiver's area, the product of width and height.
width	Answer the receiver's width.
height	Answer the receiver's height.
extent	Answer the Point receiver's width @ receiver's height.
top	Answer the position of the receiver's top horizontal line.
right	Answer the position of the receiver's right vertical line.
bottom	Answer the position of the receiver's bottom horizontal line.
left	Answer the position of the receiver's left vertical line.
origin: originPoint corner: cornerPoint	Set the points at the top left corner and the bottom right corner of the receiver.
origin: originPoint extent: extentPoint	Set the point at the top left corner of the receiver to be originPoint and set the width and height of the receiver to be extentPoint.
testing	
contains: aRectangle	Answer whether the receiver contains all Points contained by the argument, aRectangle.
containsPoint: aPoint	Answer whether the argument, aPoint, is within the receiver.
intersects: aRectangle	Answer whether the receiver contains any Point contained by the argument, aRectangle.

Figure 5.11: Rectangle Class Protocol



```
DisplayObject subclass: #DisplayT
    in                 bleNames: 'text te         ffset form
    c              Names: ''
    p
    c
```

Figure 5.13: Forms within a Form

object which contained the necessary instructions to calculate the entry coordinates of each ExternalData's display Form on the DisplayBuffer Form maintained by the InteractiveStruct (or MultiStruct) object.

5.3. The User Interface Prototype

As presented in the survey of SIMD UIDEs in Chapter 2, Smalltalk provides a Model-View-Controller (MVC) framework to develop its consistent user interfaces. This section is devoted to a detailed discussion of the MVC framework with the aim of showing how it could be modified to realize a MIMD framework.

The discussion is divided into three major parts. The first completes the description of Smalltalk's graphics kernel by presenting how input/output devices are handled. A detailed description of the MVC framework and its weaknesses are then presented. Finally, modifications to the MVC framework are presented that generalize it to a MIMD framework.

5.3.1. Smalltalk's Graphics Kernel

The previous sections have introduced most of the important classes that make up Smalltalk's graphics kernel. The major pieces missing from the discussion are graphical input and output. Input is concerned with how input devices (mouse, keyboard, tablet, etc.) are handled, as well as how user input through these devices is modeled. Output is mainly concerned with how graphical output is presented on the display device. Each of these categories is discussed below.

A three-button mouse is Smalltalk's primary input device. All other input devices are relegated to secondary status. In fact, interacting with Smalltalk without a three-button mouse can be slow and extremely difficult. This was most evident when the University of Central Florida acquired a Smalltalk site license for its Local Area Network (LAN). Most of the PCs on the LAN were not equipped with mice and users found it very difficult and cumbersome to interact with Smalltalk through keyboards, and eventually gave up their efforts in using Smalltalk.

In contrast, the Computer Science department runs Smalltalk on single SUN workstations that are equipped with three-button mice. These machines are probably the most popular within the department.

Smalltalk's (over)use of the mouse is reflected throughout the system design and implementation. The consistent interaction style imposed by Smalltalk designates the left or **red** button exclusively for selection, the middle or **yellow** button for invoking a pop-up menu of choices (of functions or methods) that are context dependent (that is, associated with the active view or portion of display within which the cursor lies), and the right or **blue** button for global functions available from any context (in Smalltalk this corresponds to window functions, such as move, frame, open, close, etc.).

The keyboard is used only when it is impossible to use the mouse, that is, for entering text. Even within text editors, the mouse is used to invoke all editor functions (such as selecting blocks, copying, pasting, etc.). The keyboard is used only to enter text within the editor.

The implementation of input devices is defined by three classes in Smalltalk: InputSensor, InputState and KeyboardEvent. Class InputSensor represents the top level interface to the input devices in the system. The system creates a global instance Sensor which represents all user input devices, that is, the mouse and keyboard. Its protocol is divided into three categories: mouse, keyboard and cursor. The mouse behavior allows for the determination of

[a] the current coordinates of the mouse (mousePoint),

[b] the state of the buttons (redButtonPressed, blueButtonPressed, yellowButtonPressed, noButtonPressed, anyButtonPressed), and

[c] the ability to wait for some user action on the mouse (waitButton, waitNoButton, and waitClickButton).

Note that InputSensor provides a **polling** mechanism to detect mouse location. Thus the cursor always follows the mouse.

Keyboard actions are event initiated. Each key press generates a keyboard event that is represented as an instance of class KeyboardEvent. Basically, each keyboard event is represented by a key character (the character representation of the key stroke), as well as a **meta state** which indicates whether a special key was also pressed (e.g., control and shift keys). Keyboard events are maintained in an event queue within class InputState which represents the state of the input devices. The keyboard behavior within InputSensor provides for

[a] determining whether the keyboard has been pressed (keyboardPressed),

[b] getting the character representation of the key that was pressed (keyboard) [note that this does not include the meta state],

[c] determining whether the control or leftshift key were pressed (ctrlDown, leftShiftDown) [note that methods are not provided to determine other meta states such as right shift and alternate keys],

[d] accessing the keyboard event itself (keyboardEvent), and

[e] manipulating the keyboard event queue (flushKeyboard, keyboardPeek).

Finally, the cursor behavior allows for accessing and setting

[a] different types of cursor shapes (e.g., up, down, left and right arrows, crosshair, hand, etc.), and

[b] the current cursor position (cursorPoint and cursorPoint:).

Thus, Smalltalk's implementation of input devices is extremely biased towards the mouse, in keeping with its consistent style of interaction. It is important to note that the mouse location can be determined by an event-based mechanism (through behavior provided in class InputState). However, since this behavior is not supported by **primitives** (similar to assembly language routines in conventional environments), the interaction becomes awkward and cumbersome. Note too that the keyboard behavior is not general, and is biased to its limited use within Smalltalk's interaction style. In trying to implement action tables (that is, tables that associate keyboard characters with mouse actions), we discovered that printable keys perform differently than non-printable keys (e.g., function and arrow keys). Associating mouse actions with printable keys allowed the cursor to move smoothly, while non-printable keys produced a jerky motion.

The display screen is represented by a single global instance Display of class DisplayScreen. DisplayScreen is a subclass of Form to distinguish it from all other forms. Class Form has already been presented in the previous section.

5.3.2. Smalltalk's Model-View-Controller User Interface Framework

The Model-View-Controller (MVC) framework is used throughout Smalltalk to develop consistent user interfaces. The MVC framework was briefly described in Chapter 2 as an example of a SIMD UIDE. This section presents the MVC framework in greater detail. To reiterate, the MVC framework comprises the following three components:

[1] the **Model** represents the application domain,

[2] the **View** displays the state of the model, and

[3] the **Controller** handles user interaction with the application and manages flow of control.

This three-way factoring of user interfaces is conceptually general enough to model any style. However, its realization in Smalltalk does not enforce this generality and is specific to the consistent user interface style of the system, resulting in a loose SIMD UIDE framework.

The discussion that follows starts by describing the MVC framework from the general conceptual viewpoint, and moves progressively toward the concrete Smalltalk implementation. It is important to point out that the MVC framework is supported by very little documentation. The only major reference is a cookbook for the MVC that is published by Xerox PARC [78]. The ensuing discussion is mainly based on actual use of the MVC implementation within Smalltalk and discussions with other Smalltalk programmers.

The MVC framework is supported by three abstract classes: Model, View and Controller. Conceptually a MVC triad contains an instance of a Model along with numerous view/controller pairs. Models maintain their associated views and controllers as dependents but do not have direct access to them. Views must be able to access their associated model to obtain the application state for display. Controllers need access to both the associated model and view. They access the model to invoke its functionality and the view to facilitate interaction with the user. A basic interaction cycle is depicted in Figure 5.14. The user interacts with the active controller to specify the action to be performed. The controller notifies the model to perform the desired action. The model performs the action which (probably) changes the application state. The model then broadcasts a message to all its dependents that it has changed. Views could access the model to inquire about the new state to update their displays.

The rest of this section is organized as follows. The first subsection presents the three abstract classes: Model, View and Controller. The next subsection presents some user interface components that aid in building Smalltalk user interfaces. The most important component is menus which form the major interaction technique used consistently throughout Smalltalk. The other tools that are described include Prompters to elicit textual input (such as names), Confirmers to elicit boolean (yes/no) input, and Lists that model a list of textual strings and form the basis for selection from a list of items. The following subsection presents complete MVC examples with the aim of showing how to build the MVC triad and more importantly to show how the MVC framework works. This section also includes a brief description of the system's main view and controller classes (StdSystemView and StdSystemContoller) which provide the basis for developing MVC triads. The final section discusses the strengths and weaknesses of the MVC framework.

5.3.2.1. The Model, View and Controller Abstract Classes

The only behavior required of models in the MVC framework is the ability to have dependents and to broadcast changes to these dependents. This behavior is implemented in class Object, the root of the entire class hierarchy. In fact, some Smalltalk implementations do not provide a separate Model class. The protocol of the Model class provides the following behavior:

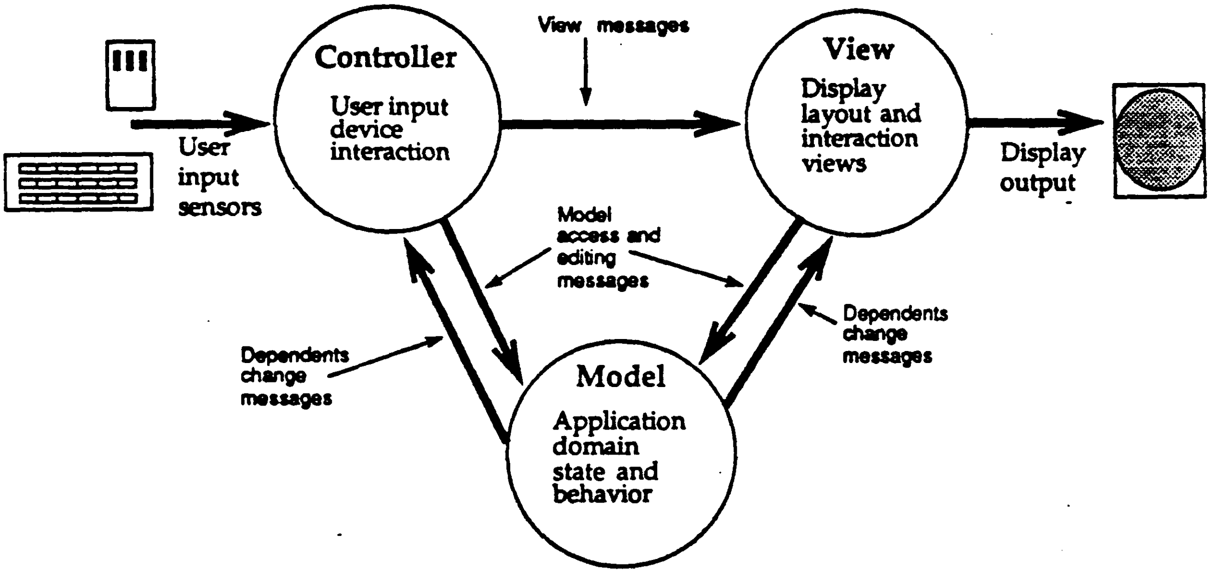

View messages
Controller
User input device interaction
View
Display layout and interaction views
User input sensors
Display output
Model access and editing messages
Dependents change messages
Dependents change messages
Model
Application domain state and behavior

Figure 5.14: Basic Model-View-Controller Interaction Cycle

[1] Manipulating dependents, that is the ability to access, add and remove dependents (dependents, addDependent:, removeDependent:) which are maintained as an OrderedCollection.

[2] Broadcasting change messages to dependents to indicate that the state of the application has changed. This behavior is defined in class Model by the protocols **changing** and **updating.** However, the methods provided within these protocols (e.g., broadcast:, broadcast:with:, changed:with:, etc.) are never used in any of the existing MVC triads within Smalltalk. One of the reasons is that none of the existing MVC frameworks use the powerful concept of associating multiple view/controller pairs with the model. All of the MVC frameworks use the **changed:aParameter** or **changed** methods provided in class Object. These methods send an update message (update:aParameter or update) to all the dependents maintained by the Model. View and Controller classes provide their own specific versions of update methods.

The View class is an abstract class that manages the display of structured pictures. A structured picture is organized as a hierarchy of views. This implies that all views, except the root or top view, have a **superView,** and can have **subViews** (maintained as an OrderedCollection). A view along with its subviews are treated as a single unit in many of class View's methods. For example, displaying a view causes its subviews to be displayed as well. Each view maintains its own coordinate system. To allow mappings between coordinate systems, each view has two transformations associated with it. The local **transformation** allows for mapping between the view and superview coordinate systems. The **displayTransformation** allows for mapping between the view and display screen coordinate systems. All transformations are instances of class WindowTransformation.

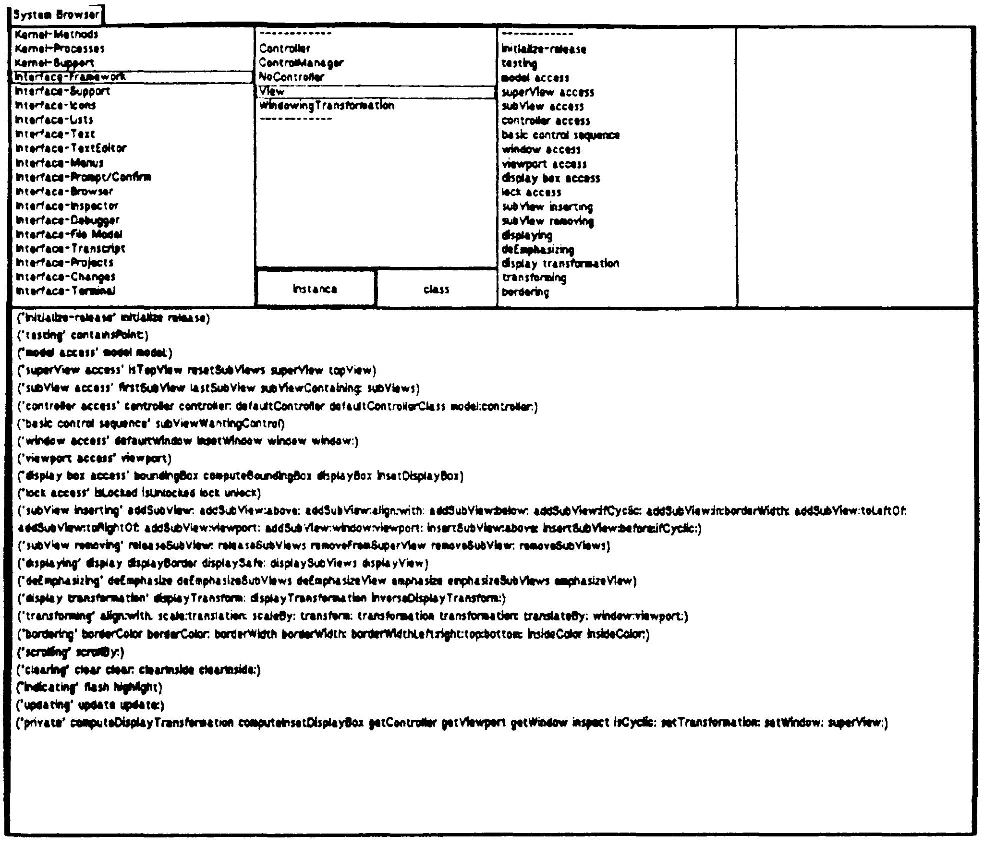
The region of the display screen occupied by the view is called its **displayBox,** an instance of class Rectangle. The displayBox can include a border which is defined by its width (borderWidth, an instance of Rectangle) and color (borderColor, an instance of Form). The region of the displayBox excluding the border is called the view's **boundingBox** (a Rectangle). Finally, a view has an associated **window** (a Rectangle) which defines the visible part of the view, and a **viewport** (a Rectangle) which defines the view's area in its superview. Note that the view does not maintain a display file or display buffer of its image. The actual image is created by subclasses of View.

A view is connected to its model and controller by actually maintaining them as instance variables. Therefore, a view has complete access to its associated model and controller, and more importantly must know about their behavior. Note that this is in sharp contrast with the model which does not require any knowledge about the behavior of its view/controller pairs, and accesses them through a well defined interface.

The abstract View class's protocol and methods are depicted in Figure 5.15. Besides operations that access various aspects of the view, the behavior of views allow for manipulating its subviews (adding and removing), displaying the view (display, deemphasizing, clearing, indicating, scrolling), transforming the view, and testing whether the view area contains a particular point.

Controllers coordinate the associated model and view with the input devices and handle interaction. Class Controller provides the abstract behavior for all controllers. It contains three instance variables: **model** to access the associated model, **view** to access the associated view, and **sensor** (an instance of InputSensor) to access the associated input device.

Before we present the behavior of controllers, it is important to understand the scheduling mechanism. A global object ScheduledControllers, an instance of class ControlManager, is the scheduler. On creation, each controller is added to ScheduledControllers list of controllers. Only one controller and its associated view are active at any given time. The scheduling of the active controller is achieved by the method searchForActiveController, which asks each controller if it wants control (implemented as method isControlWanted in Controller). The default behavior for wanting control is implemented in Controller as determining whether the associated view has the



```
('initialize-release' initialize release)
('testing' containsPoint:)
('model access' model model:)
('superView access' isTopView resetSubViews superView topView)
('subView access' firstSubView lastSubView subViewContaining: subViews)
('controller access' controller controller: defaultController defaultControllerClass model:controller:)
('basic control sequence' subViewWantingControl)
('window access' defaultWindow insetWindow window window:)
('viewport access' viewport)
('display box access' boundingBox computeBoundingBox displayBox insetDisplayBox)
('lock access' isLocked isUnlocked lock unlock)
('subView inserting' addSubView: addSubView:above: addSubView:align:with: addSubView:below: addSubView:ifCyclic: addSubView:in:borderWidth: addSubView:toLeftOf:
addSubView:toRightOf: addSubView:viewport: addSubView:window:viewport: insertSubView:above: insertSubView:before:ifCyclic:)
('subView removing' releaseSubView: releaseSubViews removeFromSuperView removeSubView: removeSubViews)
('displaying' display displayBorder displaySafe: displaySubViews displayView)
('deEmphasizing' deEmphasize deEmphasizeSubViews deEmphasizeView emphasize emphasizeSubViews emphasizeView)
('display transformation' displayTransform: displayTransformation inverseDisplayTransform:)
('transforming' align:with: scale:translation: scaleBy: transform: transformation transformation: translateBy: window:viewport:)
('bordering' borderColor borderColor: borderWidth borderWidth: borderWidthLeft:right:top:bottom: insideColor insideColor:)
('scrolling' scrollBy:)
('clearing' clear clear: clearInside clearInside:)
('indicating' flash highlight)
('updating' update update:)
('private' computeDisplayTransformation computeInsetDisplayBox getController getViewport getWindow inspect isCyclic: setTransformation: setWindow: superView:)
```

Figure 5.15: View Protocol and Methods

cursor (implemented by method viewHasCursor which is defined as **view containsPoint: Sensor cursorPoint).**

The basic control sequence protocol, defined in method **startUp,** is described by invoking the three methods **controlInitialize, controlLoop** and **controlTerminate.** The controlInitialize and controlTerminate methods are redefined in subclasses of Controller to allow for specialized initiation and termination behavior. For example, most controllers highlight their associated view on initialization and unhighlight the view on termination. The controlLoop method describes the actual controlling mechanism as follows:

```
[self isControlActive] whileTrue: [Processor yield. self controlActivity].
```

Therefore the method controlActivity is carried out as long as the controller is active. The default control activity is to pass control to the controller associated with a subview of the current view, allowing the lowest subview in the hierarchy to gain control. This method, **controlToNextLevel,** is implemented as follows:

```
| aView |
aView <-- view subViewWantingControl.
aView ~~ nil ifTrue: [aView controller startUp].
```

The subViewWantingControl, defined in class View, determines if any of its subviews wants control as follows:

```
subViews reverseDo:
  [:aSubView | aSubView controller isControlWanted ifTrue: [^aSubView]].
^nil
```

Finally, the default behavior for a controller being active (method isControlActive) is that the view has the cursor and the blue button is not pressed. Note that the blue button is used exclusively to initiate global windowing functions (such as moving, framing, collapsing) and is handled by a system controller, StdSystemController, which will be described later.

5.3.2.2. Smalltalk User Interface Components

The Smalltalk system includes a large number of classes that provide the components to rapidly build standard user interfaces. Most of these classes are subclasses of the three abstract classes: Model, View and Controller. Figures 5.16, 5.17 and 5.18 present an overview of the entire hierarchy for the three abstract classes respectively. Each Figure presents the class names and the instance variables they define. Note the convention that is used consistently to name related classes involved in a MVC triad. The View's name is the Model's name concatenated with the word View, and the controller's name is the model's name concatenated with the work Controller. For example, Switch, SwitchView and SwitchController form a MVC triad that implement buttons or switches which can be on or off.

The major component of the standard Smalltalk interface that is defined independent of the MVC abstraction are menus. All menus within the Smalltalk interface are dynamic and **pop up** at the current cursor position when activated by a specific button press on the mouse. The menu component hierarchy allow for both simple and hierarchical menus as shown in Figure 5.19.

Class PopUpMenu is the root of the menu component hierarchy, and provides the entire behavior for pop-up menus. Typically a pop-up menu consists of a list of items (strings) from which the user makes a selection. The style of interaction with pop-up menus is standardized. The pop-up menu is activated by a mouse button press. On activation the pop-up menu displays its items in a rectangular region on the screen with a selection highlighted. The user uses the mouse (with the button still depressed) to move over the desired selection. As the mouse moves over any selection, it is highlighted. When the button is released over a highlighted selection, the pop-up menu returns the selection and removes its display from the screen. Therefore class PopUpMenu bundles the model (the list of items), view (the rectangular area on the screen), and controller (the standard mouse behavior for menus) into a single class.

Model	dependents
BinaryChoice	trueAction falseAction actionTaken
Browser	organization category className meta protocol selector textMode
Debugger	context receiverInspector contextInspector shortStack sourceMap sourceCode processHandle
MethodListBrowser	methodList methodName
Explainer	class selector instance context methodText
FileModel	fileName
FileList	list myPattern isReading
HierarchicalFileList	selectionName isDirectory emptyDir myDirectory
Icon	form textRect
Inspector	object field
ContextInspector	tempNames
DictionaryInspector	ok
OrderedCollectionInspector	
StringHolder	contents isLocked
ChangeList	listName changes selectionIndex list filter removed filterList filterKey changeDict doItDict checkSystem fieldList
FillInTheBlank	actionBlock actionTaken
Project	projectWindows projectChangeSet projectTranscript projectHolder
TextCollector	entryStream
Terminal	displayProcess serialPort localEcho ignoreLF characterLimit
Switch	on onAction offAction
Button	
OneOnSwitch	connection
SyntaxError	class badText processHandle

Figure 5.16: Model Class Hierarchy

View	model controller superView subViews transformation viewport window displayTransformation insetDisplayBox borderWidth borderColor insideColor boundingBox
BinaryChoiceView	
DisplayTextView	rule mask editParagraph centered
FormMenuView	
FormView	rule mask
FormHolderView	displayedForm
IconView	iconText isReversed
ListView	list selection topDelimiter bottomDelimiter lineSpacing isEmpty emphasisOn
ChangeListView	
SelectionInListView	itemList printItems oneItem partMsg initialSelectionMsg changeMsg listMsg menuMsg
StandardSystemView	labelFrame labelText isLabelComplemented savedSubViews minimumSize maximumSize iconView iconText lastFrame cacheRefresh
AClockView	myProject date
BrowserView	
FileListView	
GClockView	cacheForm cacheBox myProject date
InspectorView	
NotifierView	contents
StringHolderView	displayContents
FillInTheBlankView	
ProjectView	
TextCollectorView	
TerminalView	
SwitchView	complemented label selector keyCharacter highlightForm arguments emphasisOn
BooleanView	
TextView	partMsg acceptMsg menuMsg
CodeView	initialSelection
OnlyWhenSelectedCodeView	selectionMsg

Figure 5.17: View Class Hierarchy

```
Controller                                 model view sensor
  BinaryChoiceController
  FormMenuController
  MouseMenuController                      redButtonMenu redButtonMessages
                                           yellowButtonMenu
                                           yellowButtonMessages blueButtonMenu
                                           blueButtonMessages
    AClockController                       clockProcess
    BitEditor                              scale squareForm color
    FormEditor                             form tool grid togglegrid mode
                                           previousTool color unNormalizedColor
                                           xgridOn ygridOn toolMenu
                                           underToolMenu
    GClockController                       clockProcess
    IconController
      ProjectIconController
    ScreenController
    ScrollController                       scrollBar marker
      ListController
        LockedListController
          ChangeListController
        SelectionInListController
      ParagraphEditor                      paragraph startBlock stopBlock
                                           beginTypeInBlock emphasisHere
                                           initialText selectionShowing
        TextEditor
          StringHolderController           isLockingOn
            ChangeController
            FillInTheBlankController
              CRFillInTheBlankController
            ProjectController
            TextCollectorController
              TerminalController
          TextController
            CodeController
              AlwaysAcceptCodeController
              OnlyWhenSelectedCodeController
    StandardSystemController               status labelForm viewForm
      NotifierController
  NoController
  SwitchController                         selector arguments cursor
```

Figure 5.18: Controller Class Hierarchy

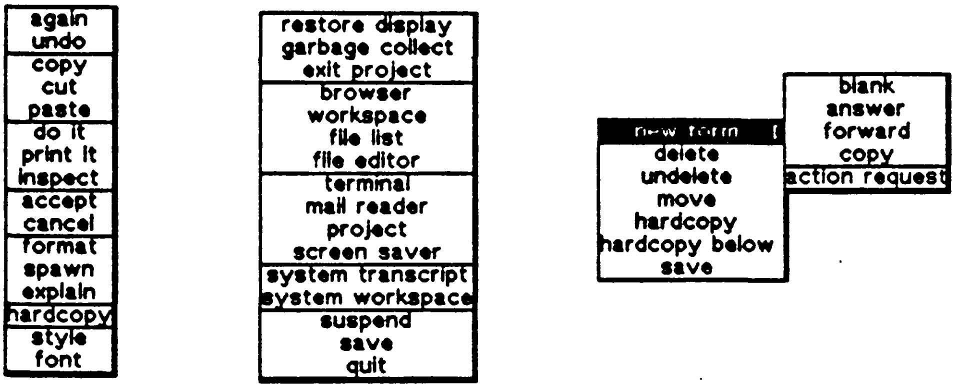


Figure 5.19: Examples of Simple and Hierarchical Menus

Class ActionMenu is a subclass of PopUpMenu that adds the ability to associate selectors with the list of items. Therefore, when the user makes a selection from the list of items, its associated selector is returned. This allows methods to be associated with choices, and can be performed immediately when its corresponding item is selected. Class HierarchicalMenu completes the menu component hierarchy. It allows entire pop-up menus to be associated with the list of items.

The use of pop-up menus within the MVC triad is achieved by class MouseMenuController. This class provides the behavior for associating pop-up menus along with a corresponding set of messages to be connected to the three mouse buttons. The messages correspond to methods to be performed when the corresponding item is chosen. The control activity behavior is defined as follows:

```
sensor redButtonPressed & viewHasCursor    ifTrue: [^self redButtonActivity].
sensor yellowButtonPressed & viewHasCursor ifTrue: [^self yellowButtonActivity].
sensor blueButtonPressed & viewHasCursor   ifTrue: [^self blueButtonActivity].
```

For example, the blueButtonActivity method is defined as follows:

```
| index |
blueButtonMenu ~~ nil
  ifTrue: [index <-- blueButtonMenu startUp.
        index ~= 0 ifTrue: [self menuMessageReceiver
                                perform: (blueButtonMessages at: index)]]
  ifFalse: [super controlActivity].
```

This method checks to see if a menu has been attached to the blue button. If not, it calls its parent's control activity. Otherwise, the pop-up menu associated with the blue button takes control (by sending it the message startUp), and the user makes a selection. If a selection is made then the receiver of the menu is asked to perform the corresponding method stored in the message list for the blue button.

Prompters and confirmers are MVC triads that are used to elicit simple user input. They are created (and take control immediately) by providing the prompt string and the action to be taken on selection. They return a string value and a boolean value, respectively. Figure 5.20

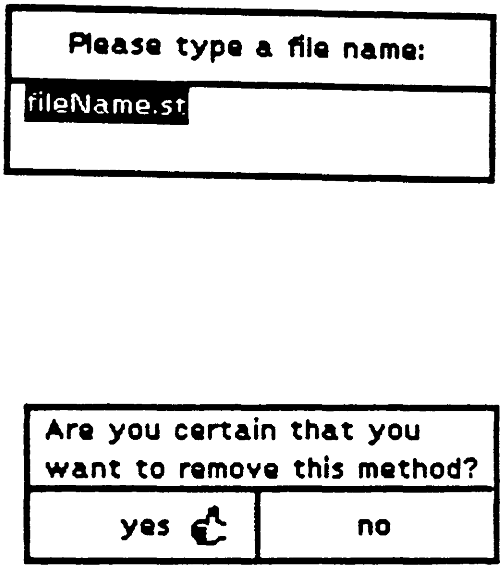
shows examples of prompters and confirmers.

Another important MVC triad component provides the ability to make a selection from a list of strings. This is the MVC counterpart of the pop-up menus presented earlier, and can be viewed (sic!) as providing facilities for static (always visible) menus. They are implemented by the triad classes List, ListView and ListController, and are used extensively throughout Smalltalk. Figure 5.21 shows Smalltalk's system browser which allows the user to browse (and manipulate) the class hierarchy. The four views at the top of the browser are ListViews. The bottom view shows the code associated with the method redButtonActivity (defined in class ListController) which describes the behavior for selecting an item from the list. Note that the code does not include scrolling behavior. The scroll bar appears automatically when the cursor moves into the view.

So how is scrolling achieved? Controllers of views that can be scrolled are defined as subclasses of class ScrollController (a subclass of MouseMenuController) which provides Smalltalk's standardized scrolling behavior. Whenever a scrollable view's controller is activated, its controlInitialize method displays a scroll bar at the left of the view. The scroll bar represents the length of the information in the view. The marker or slider inside the scroll bar depicts the amount of the information that is currently shown relative to the total length. ScrollController provides the behavior for scrolling by taking control whenever the cursor is inside the scroll bar area, as defined in its controlActivity method:

```
self scrollBarContainsCursor
  ifTrue: [self scroll]
  ifFalse: [super controlActivity].
```

Scrolling is achieved either by predefined (up/down) increments, or by grabbing the marker and moving it to the desired position. The calculation of the marker's extent is done by dividing the associated view's window (the portion being shown on the screen) height by its boundingBox (the total area of the view) height. Note that no provision is made in Smalltalk for horizontal scrolling.

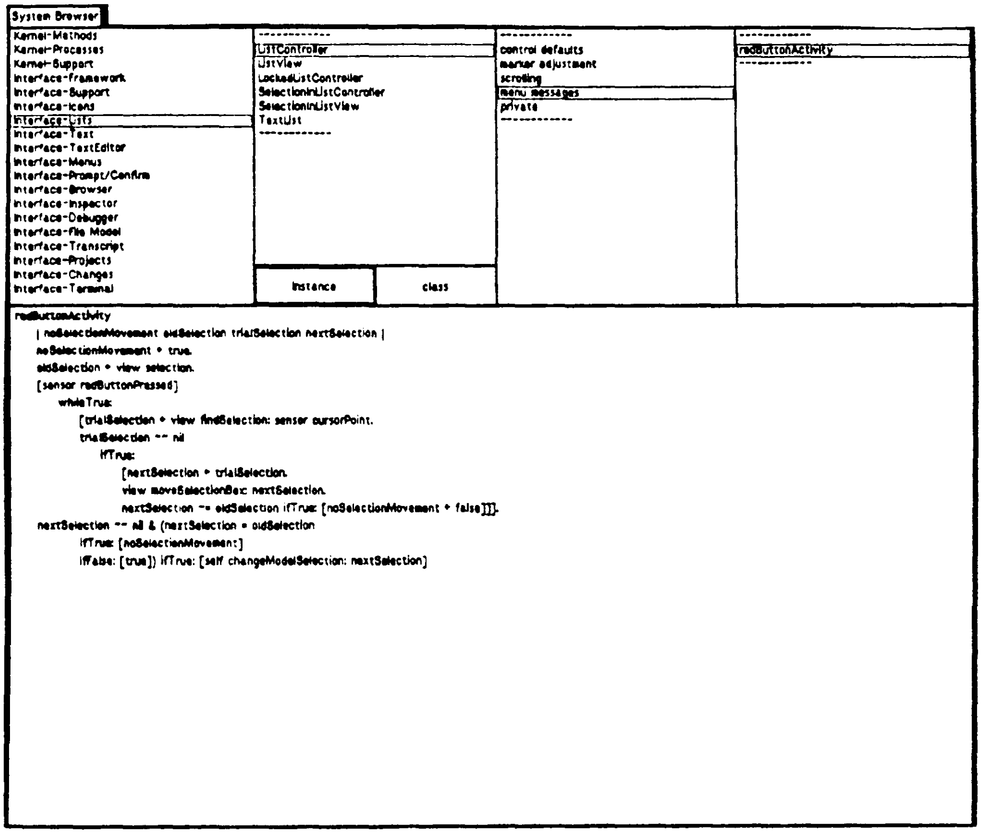


Please type a file name:

fileName.st

Are you certain that you want to remove this method?

Figure 5.20: Examples of a Prompter and a Confirmer



```
redButtonActivity
    | noSelectionMovement oldSelection trialSelection nextSelection |
    noSelectionMovement ← true.
    oldSelection ← view selection.
    [sensor redButtonPressed]
        whileTrue:
            [trialSelection ← view findSelection: sensor cursorPoint.
            trialSelection ~~ nil
                ifTrue:
                    [nextSelection ← trialSelection.
                    view moveSelectionBox: nextSelection.
                    nextSelection ~= oldSelection ifTrue: [noSelectionMovement ← false]]].
    nextSelection ~~ nil & (nextSelection = oldSelection
            ifTrue: [noSelectionMovement]
            ifFalse: [true]) ifTrue: [self changeModelSelection: nextSelection]
```

Figure 5.21: The System Browser

5.3.2.3. MVC Examples

The root of any MVC triad is an instance of class StdSystemView, which along with its associated controller, an instance of class StdSystemController (a subclass of MouseMenuController), define the behavior of the window system. StdSystemView provides an (empty) window with a label on top. MVC triads are created as subviews of a StdSystemView. StdSystemView contains the behavior for manipulating the label, the view as a whole (for example, moving, reframing, and collapsing), and the size of the view on the display. Its associated StdSystemController defines the default behavior for invoking windowing functions associated with the blue button of the mouse. The specifics of the behaviors of both these system classes will be described in the following examples.

The general (undocumented) rule for building a MVC triplet is to build the model, and then the view, and then the controller. The view is built before the controller to allow the functionality of the triad to be tested immediately. The following presents two complete examples of MVC triads.

The Counter example is the simplest demonstration of a full MVC triad. Very simply, the model is a numerical value that can be incremented/decremented, the view shows the value of the counter, and the controller implements a pop-up menu allowing the user to increment or decrement the counter's value. The model for the counter is shown in Figure 5.22 and is self explanatory. The CounterView class is defined as a subclass of View and is responsible for displaying the state of its model, Counter. The definition of CounterView is shown in Figure 5.23. It includes a method displayView for displaying the value maintained in its associated Counter model. Note the update: method which is used to update the view when the model changes. The defaultControllerClass defines the type of controller for the view and is automatically created when the view is created. The CounterController class is depicted in Figure 5.24 and its behavior is self explanatory. The only aspect missing from the definition is how the MVC triplet is created. This is shown in Figure 5.25 and is implemented as a class method in CounterView.

```
Model subclass: #Counter
        instanceVariableNames: 'value '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Demo-Counter'


Counter methodsFor: 'initialize-release'
initialize
        "Set the initial value to 0."
        self value: 0

Counter methodsFor: 'accessing'
value
        "Answer the current value of the receiver."
        ↑value

value: aNumber
        "Initialize the counter to value aNumber."
        value ← aNumber.
        self changed            "to update displayed value"

Counter methodsFor: 'operations'
decrement
        "Subtract 1 from the value of the counter."
        self value: value -1

increment
        "Add 1 to the value of the counter."
        self value: value + 1

Counter class methodsFor: 'instance creation'
new
        "Answer an initialized instance of the receiver."
        ↑super new initialize     "return a new instance of the receiver"
```

Figure 5.22: The Counter Model

```
View subclass: #CounterView
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Demo-Counter'

CounterView methodsFor: 'displaying'
displayView
        "Display the value of the model in the receiver's view."
        | box pos displayText |
        box ← self insetDisplayBox.        "get the view's rectangular area for displaying"

                                           "Position the text at the left side of the area,
                                                   1/3 of the way down"
        pos ← box origin + (4 @ (box extent y / 3)).
                                           "Concatenate the components of the output
                                                   string and display them"
        displayText ← ('value: ', self model value printString) asDisplayText.

        displayText displayAt: pos

CounterView methodsFor: 'updating'
update: aParameter
        "Simply redisplay everything."
        self display

CounterView methodsFor: 'controller access'
defaultControllerClass
        "Answer the class of a typically useful controller."
        ↑CounterController
```

Figure 5.23: The Counter View

```
MouseMenuController subclass: #CounterController
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Demo-Counter'

CounterController methodsFor: 'initialize-release'
initialize
        "Initialize a menu of commands for changing the value of the model. "
        super initialize.
        self yellowButtonMenu: (PopUpMenu labels: 'Increment\Decrement' withCRs)
            yellowButtonMessages: #(increment decrement)
CounterController methodsFor: 'menu messages'
decrement
        "Subtract 1 from the value of the counter."
        self model decrement

increment
        "Add 1 to the value of the counter."
        self model increment

CounterController methodsFor: 'control defaults'
isControlActive
        "Take control when the blue button is not pressed."
        ↑super isControlActive & sensor blueButtonPressed not
```

Figure 5.24: The Counter Controller

```
CounterView class methodsFor: 'instance creation '
open
        "Open a view for a new counter."
        "select and execute this comment to test this method"
                "CounterView open."

        | aCounterView topView |


                                                "create the counter display view"
        aCounterView ← CounterView new                  "a new CounterView instance"
                model: Counter new.                      "with a Counter as its model"

        aCounterView borderWidth: 2.                     "give it a borderWidth"
        aCounterView insideColor: Form white.            "and white insides"


                                                "the top-level view"
        topView ← StandardSystemView new                "a new system window"
                label: 'Counter '.                       "labelled 'Counter' "

        topView minimumSize: 80@40.                      "at least this big"
                                                "add the counterView as a subView"
        topView addSubView: aCounterView.
                                                "start up the controller"
        topView controller open
```

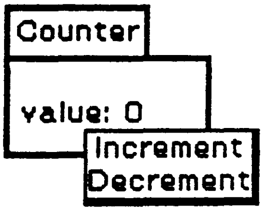


Figure 5.25: Creating the Counter MVC

Figure 5.26 shows the interfaces between the MVC components for this example.

The basic steps of the MVC triad creation is as follows [note that the ordering of the steps is not relevant].

[1] Create the views of the MVC triad with their corresponding model.

[2] Create a top view, an instance of StdSystemView.

[3] Attach each view of the MVC triad as a subview of the top view, defining the layout of these subviews within the top view.

[4] Perform any initializations.

[5] Schedule the top view's controller.

The last step invokes the **open** method defined in StdSystemController which allows the user to define the area for the view, displays the top view (and its corresponding subviews), and then assigns itself as the active controller with the control manager, ScheduledControllers. It is important to note that the MVC framework does not enforce where the MVC creation should take place. In fact, the Smalltalk system has examples of the creation process done by the model (e.g., FileList's openOnPattern:), the View (e.g., ProjectView's open:) and the controller (e.g., FormEditor's createOnForm:). Conceptually, the creation message should be carried out within the view, because it has direct access to both the model and the controller, and more importantly because the creation message mainly concerns setting up the view within a top StdSystemView.

The next example is a simple editor to manipulate the contents of a list of strings. Such an editor could be used, for example, to maintain a list of message names for browsing. The MVC triad for such an editor is implemented by classes ListHolder, ListHolderView and ListHolderController. The ListHolder model contains an ordered collection that represents the list. The functionality of the editor is defined by standard editing facilities such as undo, edit, copy, cut and paste. Many of these functions implicitly use a Macintosh style clipboard. The class definitions

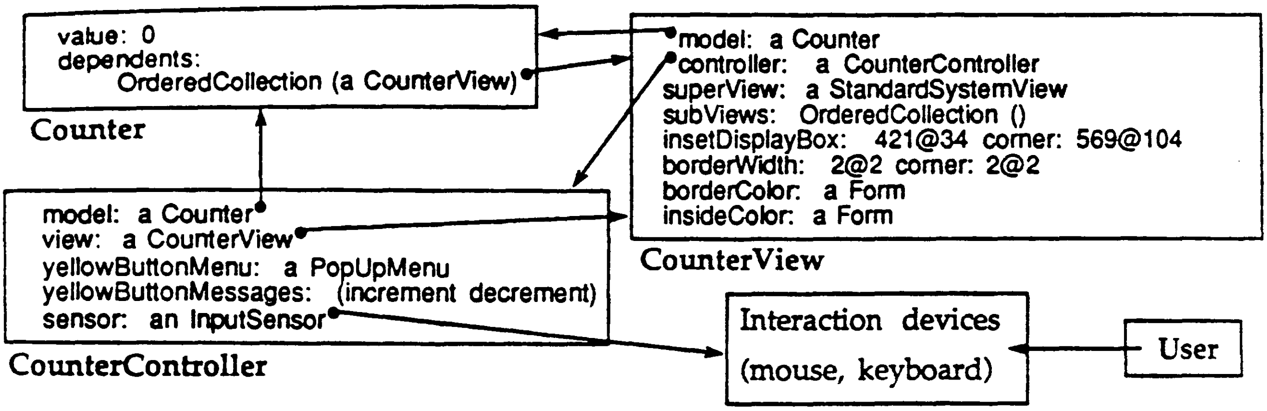

value: 0
dependents:
OrderedCollection (a CounterView)
Counter
model: a Counter
controller: a CounterController
superView: a StandardSystemView
subViews: OrderedCollection ()
insetDisplayBox: 421@34 corner: 569@104
borderWidth: 2@2 corner: 2@2
borderColor: a Form
insideColor: a Form
CounterView
model: a Counter
view: a CounterView
yellowButtonMenu: a PopUpMenu
yellowButtonMessages: (increment decrement)
sensor: an InputSensor
CounterController
Interaction devices
(mouse, keyboard)
User

Figure 5.26: Interfaces of the Counter MVC

for the model, view and controller are presented in Appendix C.

5.3.2.4. Weaknesses of the MVC Framework

The major strength of the MVC framework is that it provides a large number of prefabricated components that allow designers to very rapidly create standardized Smalltalk user interfaces. It is also extremely flexible (in the confines of building consistent interfaces) because any of the classes can be modified or extended very easily. However, its weaknesses far outshadow its strengths. The rest of this section discusses its weaknesses. The discussion starts with the weaknesses of the MVC framework in general and gradually describes specific weaknesses.

The major drawback of using the MVC framework is that it requires intimate knowledge of the structure and implementation of all its components. As was pointed out in the introduction to this chapter, this makes the MVC framework a white-box framework. This drawback is made worse by the lack of decent documentation. This implies a large learning curve, which is exacerbated by the numerous undocumented and non obvious protocols, guidelines and subtle rules that are necessary to gain a complete understanding of the MVC framework. Note that the recent OOPSLA conference included an entire tutorial dedicated to MVC programming, supporting the difficulty of mastering it.

As a white-box framework, the MVC is poorly organized and designed. Kent Beck [107] puts it aptly:

> **The people who wrote it had an incomplete understanding of what their code was supposed to do and where it would fit with the rest of the system.**

Conceptually, the MVC framework is a powerful concept. However, its realization and extension is extremely weak.

The major criteria for a well-designed white-box framework is as follows [95].

[1] The major components of the design are represented within the framework as abstract classes which have well defined interfaces between them.

[2] The entire hierarchy that implements all aspects of the framework should be narrow and deep, that is, the abstract classes should be the roots of the entire class hierarchy.

[3] All the subclasses should adhere to the major principles of the overall design abstraction.

The following shows that the MVC framework does not meet any of the above criteria.

The Model abstract class is the weakest link in the design. It does not provide any behavior or structure for implementing applications. The only behavior that is provided is for maintaining the associated view/controller pairs as dependents which is used to broadcast application state changes. This behavior, however, defines a particular aspect (change management) of the interface between the model and its associated view/controller pairs.

The View abstract class provides a well defined behavior for its interaction with the screen and its subviews. However, it does not define any behavior or structure for its interaction with the associated model. The manner in which the view interacts with the model is defined in subclasses of View, thereby losing the generality of the design abstraction. More importantly, class View does not define any interface between itself and the corresponding model. This interface is achieved by maintaining the model object itself as an instance variable. This is a poorly designed interface between components of a framework, and is the major reason why each view subclass is mainly concerned with defining this interface with the model. Note that the subclasses of View are implementing an application behavior, that is, how the external representation of its state is created. These subclasses require an intimate knowledge of the actual implementation of the model which is highly undesirable in abstract components. The model, on the other hand, requires no knowledge of its corresponding views.

The interface between the view and its controller is much better. The defaultControllerClass method is a good example of automatic connection between components, as it allows a view to instantiate an appropriate controller to control itself. Views only provide behavior to return their controllers and never actually access the controller itself. However, the view actually maintains its controller object as an instance variable and therefore could potentially access the internals of the controller. This is again an example of a poorly designed interface between components.

The same philosophy is also used in defining the interfaces between the controller and its associated view and model. Once again, controllers are forced to have intimate knowledge of structure and implementation of the view and model. The major weakness of the Controller class (and of the entire MVC framework in general) is that the three-button mouse is bound as the interaction device very early in the design. This does not provide for any other devices to be used.

The entire class hierarchy that implements the user interface components basically falls under the three abstract MVC classes. The major exception is the implementation of pop-up menus. Class PopUpMenu is a subclass of Object, and includes its own view and controller. It is interesting to note that static menus, implemented by the MVC triplet List, ListView and ListController, present the same functionality as pop-up menus. The only difference is that one is static (always visible) and the other dynamic. Thus, their only difference is in the viewing facilities. Since displaying a view causes all its subviews to be displayed, class PopUpMenu is defined separately only because of the inability of class View to dynamically display some of its subviews.

The major goal of a framework is to divide a design into a specific number of components which carry out well defined functions within the design. Therefore the framework should enforce where each function of the overall design should be performed. The conceptual functionality of the abstract MVC classes is not enforced in their implementations. This non-enforcement of functionality within each component is probably the gravest problem with the effectiveness of the MVC framework. This major problem with the MVC framework is best described by Sam Adams [108]:

> A simple analogy of MVC is a computer system where the Model is a database manager, the View is a terminal display, and the Controller is the application program running the whole show. While this analogy is useful at times to explain the basic divisions of labor in MVC, it presents a poor example of the fact that in Smalltalk, the "application program" is

> typically scattered over the behaviors of several objects.
>
> And that is exactly the reason why many programmers have a difficult time learning how to use an MVC. I am constantly being asked the question, "which methods should I put in the View and which should I put in the Controller or Model?".
>
> In my experience I have observed that only the most simple applications of MVC's in Smalltalk applications adhere strictly to the general guidelines. In fact, in the standard image of Smalltalk-80, there are several examples where a View has a built-in Model or has no Controller. ......

The previous subsections have presented numerous examples of the inconsistency of the MVC framework. For example, the MVC triad can be created in any component of the MVC triplet. Another important inconsistency is the scrolling behavior for views which is defined within class ScrollController. Scrolling exemplifies the ad-hoc manner in which the MVC framework has evolved. Object-oriented framework evolution should be reflected within the root abstract classes of the framework. It should not be achieved by destroying the design abstraction at lower levels in the hierarchy.

We end this section just as we began it, on a positive note. There exist examples of some black-box frameworks within the MVC framework implementation, known as **pluggable views.** Experience with the MVC implementation revealed that the views and controllers of MVC triads dealing with lists, text and code were essentially similar. Pluggable views factor out this common behavior and allow them to be specified as parameters in a creation message. The major pluggable views within Smalltalk-80 are SelectionInListView, TextView and CodeView.

SelectionInListView is a black-box framework built from the MVC triad List, ListView and ListController, which were presented earlier as (basically) an implementation of static menus. The SelectionInListView provide a parameterized instance creation method **on:aspect:change:list:menu:initialSelection:** that automatically creates an instance of a static menu. For example, the list of classes presented by the system browser (as the second view on the top of Figure 5.22) can be created as follows:

```
ClassListView <-- SelectionInListView  "an instance of SelectionInListView"
            on: aBrowser        "model of the SelectionInListView"
            aspect: #className   "message to get the selected item"
```

```
change: #className:  "message sent on item selection"
list: #classList    "message sent to generate list"
menu: #classMenu    "message sent to get menu"
initialSelection:
    #className    "message sent to get initial selection"
```

The messages indicated by #<symbol> is implemented by the corresponding model aBrowser. Figure 5.27 shows the classMenu method (in the lower view) which provides functions for manipulating classes as shown in the pop-up menu in the second view on the top.

Appendix D presents a complete example of the use of SelectionInListView for creating class Notebook which manages a two-level topic hierarchy, each with associated text. Some examples of notebooks might be a calendar organized by month and day or a university course catalogue organized by department and course title.

5.4. Generalizing the MVC Framework

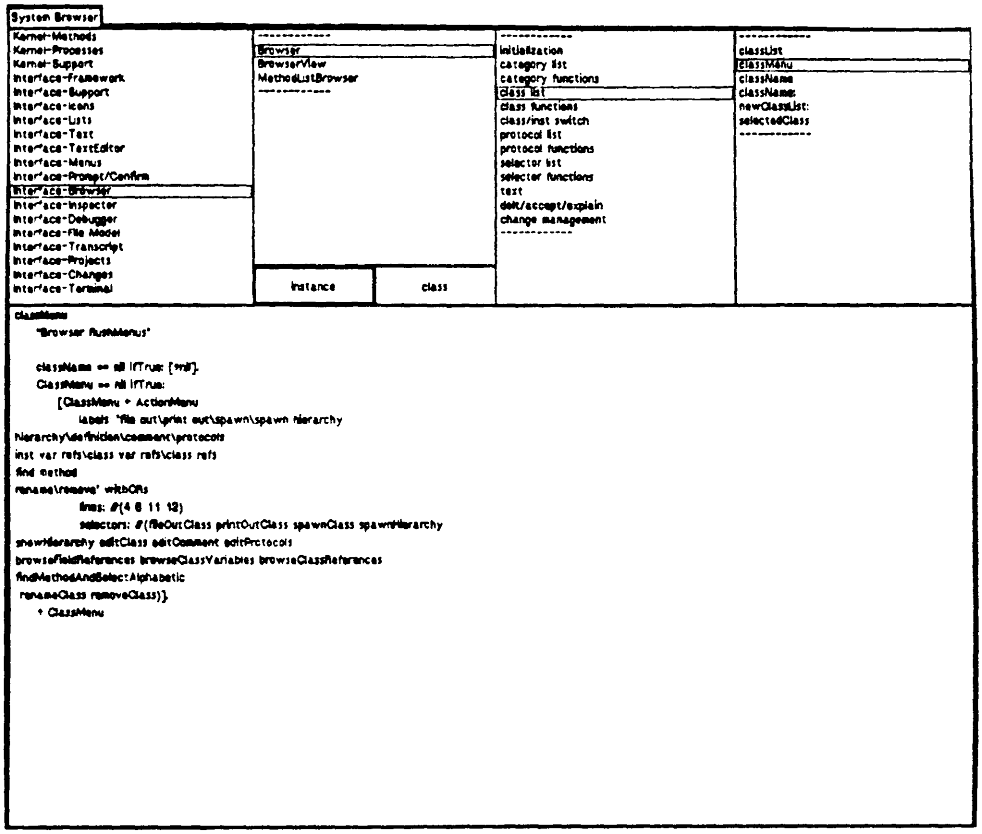
This section discusses modifications to the MVC framework in order to

[a] make it into a generalized white-box framework, and

[b] push it towards a flexible MIMD framework.

The modifications that are suggested are aimed at the basic components for creating user interfaces within Smalltalk. That is, the MVC abstract triplet Model, View and Controller. The discussion will also focus on important features of user interfaces such as menu, scrolling, support for input/output devices, and scheduling (and managing) controllers.

The major goal of these modifications is to ensure that the conceptual abstract design of the MVC framework is maintained in its implementation. The first step in this generalization is to determine the exact behavior of each of the major components in the MVC framework. The next step builds well-defined interfaces between these components. The final step ensures that the protocol (or behavior) defined in the root abstract classes is followed by all its subclasses. Therefore, none of the subclasses should be allowed to break or combine the behavior of the abstraction. However, the design should allow black-box frameworks to be developed to



```
classMenu
    "Browser flushMenus"

    className == nil ifTrue: [↑nil].
    ClassMenu == nil ifTrue:
        [ClassMenu ← ActionMenu
            labels: 'file out\print out\spawn\spawn hierarchy
hierarchy\definition\comment\protocols
inst var refs\class var refs\class refs
find method
rename\remove' withCRs
            lines: #(4 8 11 12)
            selectors: #(fileOutClass printOutClass spawnClass spawnHierarchy
showHierarchy editClass editComment editProtocols
browseFieldReferences browseClassVariables browseClassReferences
findMethodAndSelectAlphabetic
 renameClass removeClass)].
    ↑ ClassMenu
```

Figure 5.27: The classMenu Method

facilitate the automatic generation of standard components. For example, the design should allow the development of classes such as SelectionInList (presented in the previous section) which automatically creates the view/controller pair for selecting from a list of items.

The discussion on generalizing the MVC framework is presented from the perspective of its three major components, that is the Model, View and Controller. The following subsection presents an overview of the modifications that are suggested. The final subsection summarizes the generalization process by comparing it to the MIMD UIDE design, and draws some conclusions about the development of object-oriented user interface development environments.

5.4.1. A General MVC Framework

Conceptually, the MVC framework separates the application (Model) from its user interface (View/Controller pair). In its current implementation, the Model is the weakest component. This weakness forces the associated View/Controller pair to be involved in application behavior, thereby tightly binding the user interface with the application. To achieve a general MVC framework, it is necessary to ensure a clean separation between the Model and its View/Controller pair.

This separation forces all application behavior to be defined within the Model component. The major behavior of the Model that is involved in user interaction is the external representation(s) of its internal state. Thus, the Model is modified to generate its external representation on a Form. This Form (or an array of Forms for multiple external representations) defines the interface between the Model and its View. The View accesses its application Form to present to the user. Whenever the application's state changes, it regenerates its external representation(s) and informs its View(s) to update themselves. This causes the View to access the application's Form for redisplay to the user.

The other important behavior of the Model is its functionality. Since the Model is the only component in the MVC triplet that has knowledge about application functionality, it should create the interface to access its functionality. This functionality is interfaced to the outside world through two objects: a list of items and a corresponding list of selectors (hereafter referred to as the application's **Item/selector** pair). The list of items are strings corresponding to names of functions provided by the application, and the list of selectors are the actual application methods to be invoked when its corresponding item is accessed.

An application is presented to the user within a single window on the screen. This window contains a set of subviews corresponding to external representations presented by the application. The top level view behavior is similar to that described for StdSystemView. That is, it provides an empty rectangular region with a label, handles windowing functions (such as move, frame, close) and contains the top level controller for the entire application. All other views are associated with the application through a Form that represents the application's state. The main behavior of such application view allows for manipulation of this Form, including scrolling and zooming.

In keeping with the design of maintaining View/Controller pairs, it is appropriate to associate the application's functionality (that is, its item/selector pair) with the View. This results in well-defined interfaces between the three components. The View is the only component that communicates with the application, accessing it through a well-defined interface (the Form that defines the application's external representation and the item/selector pair that defines the application's functionality). The View is also the only component that communicates with the Controller.

The Model's item/selector pair, maintained by the View, forms the basis for generating menus. Thus, menus are associated with Views. Menus are also represented by an MVC triplet: the Model is the item/selector pair, the View is the rectangular region displayed on the screen to present the menu to the user, and the Controller allows the user to a make a selection. The selection is passed by the Controller to its associated View, which in turn invokes the appropriate action in the Model (corresponding to the entry in the selector). The Model performs the associated action, creates its new external representation and informs the View that it has changed. The View updates itself by accessing the Model's Form and redisplaying it. Note that menu views are registered as subviews of the associated view. Subviews can be designated as visible or invisible, to allow for both static and dynamic views. Besides being a good model for static and pop-up menus, this feature allows any view to be dynamically displayed when activated. This is helpful in allowing the user to manipulate the information that is presented by the application. Note that such a feature also generalizes scrolling, supporting both Macintosh-like static scroll bars and Smalltalk-like dynamic scroll bars. The actual scrolling behavior is standardized in class View, since scrolling is always achieved by moving a window on the Form maintained by the View.

The Controller component needs modification to reflect the modifications in the other two components. There exists a single instance of a scheduler or control manager. Its main behavior is to determine which application wants control. This can be achieved either by asking each top level view whether the cursor is in their associated view's area, or by registering all top level controllers with the scheduler (and using the method isControlWanted to determine which controller wants control). Note that the default behavior for isControlWanted in the current MVC implementation is viewHasCursor. Therefore, the former approach is preferable. If the cursor lies outside all application windows, a screen controller is given control to allow global Smalltalk functions (such as save, quit, refresh, etc.) to be accessed. If the cursor lies within an application window, the corresponding view is activated. The default behavior for activating windows is to give control to its top level controller. When the user moves the cursor outside the confines of the current active application window, control is returned to the scheduler.

This view-oriented control scheduling is followed throughout the generalization. The top level controller determines if any of its subviews want control by invoking the isActivation method. Note that this message is sent to all subviews, including those that are not visible. This is to allow dynamically activated views to take control on a button or key press. Menu, scroll and editing controllers specialize this general behavior to suit their particular functionality.

Finally, each controller should indicate the input devices that are needed. Ideally, the system should provide low level behavior for all input devices (mouse and keyboard), and then allow the class InputSensor to be used to provide access to these devices. Thus, instances of InputSensor could provide access to only the mouse or keyboard, or to both. This allows interaction to be based on any (combination of) input device(s). Note that only the input device(s) defined by the instance of InputSensor are active, all others are deactivated.

5.4.2. Remarks on Generalizing the MVC Framework

It is interesting to compare the current MVC implementation with the MIMD UIDE developed in the previous Chapters. The major weakness of the MVC is that it fails to cleanly separate the application from the user interface. This is mainly due to the fact that the Model component does not define any structure or behavior for the application. In contrast, the MIMD UIDE imposes a strict separation, by developing the necessary models that define the exact behavior of the application. Interactive data structures model a well-defined interface between the application and the view presented to the user. The operations defined for interactive data structures provide for smooth translations between the view and the internal state of the application. Furthermore, function interaction and service interface objects model a well-defined interface between the controller and the application's functionality.

Furthermore, the Controller component of the MVC implementation is very specialized to the input device and interaction style imposed by Smalltalk. The MIMD UIDE develops a layered approach to the controller through the input device, action table, input technique, mapper and interaction task classes. This layered approach facilitates easy modification at every level, and more importantly supports a wide range of styles and techniques.

The modifications that were suggested in the previous section were basically to augment the MVC framework with the strengths of the MIMD UIDE. We would like to point out that some of these modifications (with the notable exception of modifications to the Model) have been incorporated into a version of Smalltalk that was ported to the Macintosh. We are not sure whether these modifications to the MVC framework in Smalltalk/V-MAC [109] were made to reorganize a weak framework, or to take advantage of the powerful user interface toolbox [110] provided by Macintosh. However, this reorganization of the MVC framework brings out an important problem with current object-oriented development environments: the lack of tools to reorganize the class hierarchy [95]. When a weakness is encountered in a framework, it should be strengthened in reorganizations/modifications at the highest abstract class level. The lack of reorganization tools filters weaknesses to the lower concrete classes thereby making the problems worse. Currently, the only way to enhance a framework is to totally reorganize the entire class hierarchy, which is infeasible due to time and economic considerations.

CHAPTER 6

CONCLUSIONS AND FUTURE RESEARCH

A large percentage of software engineering is devoted to the development of interactive software systems. Interactive software systems are comprised of two components: the application component defines the system's functionality and the user interface component defines the behavior of the system. It is widely accepted that the user interface component is more important since it determines the effectiveness of the interactive system. User interface software engineering research can be divided into two broad categories. The first is concerned with developing new technologies for interactive styles, techniques and devices. The second is concerned with building User Interface Development Environments (UIDEs) to aid in the construction of user interfaces. The research presented in this dissertation falls into the latter category.

Current research in UIDEs focuses on models, techniques and tools that allow the system designer to rapidly develop user interfaces for interactive applications. A major contribution of this research is that it develops a new perspective of UIDEs. This perspective views the end user of interactive systems as the only person capable of determining what constitutes an effective user interface. Thus, the emphasis of developing the user interface has shifted from the system designer to the end user. The main goal of this research has been to develop UIDEs that provide the end user with facilities to construct user interfaces, similar to those provided to system designers by currently available UIDEs. That is, end users should be able to customize every aspect of the user interface for any interactive application.

Besides maximizing user productivity and therefore system effectiveness, such a perspective has the following advantages.

[a] A UIDE that supports interface construction by the end user, automatically supports construction by the system designer as well. Thus, system designers can initially create a default user interface that defines their interpretation of the most effective way of using the interactive system's functionality. The end user can use this default interface to gain an understanding of the interactive system, and then modify it to suit his/her own personal behavior to maximize productivity.

[b] Since the user interface can be modified at any time, it can evolve incrementally with the user's experience. This is probably the most important advantage of end user customizable user interfaces. Current interactive systems present the same interface throughout their lifetime. After an initial gain in end user productivity, such an approach does not allow the user interface to increase productivity continuously by taking advantage of the growing expertise of end users.

[c] A non-changing interface is prone to technological obsolescence. Thus, any major shift in interaction styles, techniques and devices would render the entire interactive system obsolete. Furthermore, this would require a costly rebuilding of the user interface component for all the interactive systems built on an obsolete user interface style. More importantly, such an approach introduces these technological advancements at a very large granularity. That is, the new interactive system presents a completely different behavior from the previous system, and therefore requires users to relearn behavior for accessing functionality that is well understood. This retraining and relearning is extremely expensive and usually hinders the use of newer technologies. Modern interactive software engineering has failed to adequately address the problems associated with technological change. However, a user modifiable interface can accommodate the introduction of newer technologies without affecting user productivity, and more importantly does not render the entire system obsolete. Thus, users can incorporate these new technologies incrementally into their user interfaces according to their own pace of learning.

In the ever changing world of interaction technology, it is imperative to build UIDEs that support continuous modification of every aspect of user interfaces. This research is the first attempt at developing such a user-modifiable UIDE.

The rest of this Chapter is organized as follows. Section 6.1 presents the specific contributions of this research and summarizes results. One of the most exciting results of this research is that it has provided a platform for future interactive software engineering research. However, to be effective as a platform, it is necessary to build a fully operational and efficient UIDE. Section 6.2 outlines immediate future plans for building a complete user-modifiable UIDE. Finally, section 6.3 discusses how the experiences in developing such a UIDE can be applied to generating entire interactive tools and ultimately environments.

6.1. Research Summary and Contributions

This dissertation developed a classification scheme for UIDEs, based on the multiplicity of user interfaces and applications that can be supported. This is the first attempt at developing such a classification. Classification schemes are important in any research area as they provide the basis for contrasting and evaluating different approaches and models. The classification scheme for UIDEs has been helpful in identifying the characteristics of UIDEs, and more importantly has aided in understanding their strengths and weaknesses. Most of the UIDEs that have been developed to date fall in the SISD, SIMD, and MISD **[S= Single, I= user Interface(s), M= Multiple, D= application Domain(s)]** classes, and were surveyed in Chapter 2.

Current user interface development efforts are directed at the popular SIMD class. SIMD UIDEs support the generation of a single consistent interface style for many applications. The many disadvantages of such an approach has been presented in the introduction to this Chapter. The major goal of this research was to develop a MIMD UIDE, capable of generating multiple user interfaces for many applications.

It is widely accepted that a complete separation of the user interface from the application is necessary to support the development of flexible UIDEs. This dissertation has shown that

UIDEs in the SISD, SIMD and MISD classes do not achieve a clean separation, which is the major reason for their inflexibility. In Chapter 3, models were developed for both the application as well as the user interface components that provide a clean and well defined separation.

Interactive applications were modeled as general editors capable of manipulating an arbitrary 2-dimensional (structured) picture. The internal state of the picture, along with its external representations, are maintained within **interactive data structures.** These external representations are interfaced to the user by the user interface component, and form a well-defined interface between the two components. Furthermore, the model generalizes the functionality of the editor to allow the user interface component to be solely responsible for interaction. The model developed for the user interface component generalizes both the presentation of information to the user, as well as interaction with the application. The layered interaction framework allows for modification at a very fine level of granularity. Thus, the model facilitates the development of customizable interfaces.

The design of the MIMD UIDE presented in Chapter 4 provides a robust set of object-oriented classes. These object-oriented classes form the basis for an MIMD UIDE that is easily extensible to accommodate an ever changing technological environment.

Finally, Chapter 5 presented the experiences of implementing the MIMD UIDE prototype by generalizing Smalltalk's Model-View-Controller (MVC) user interface framework. The MVC framework, introduced in Chapter 2 as an example of an SIMD UIDE, is used within Smalltalk to develop consistent interfaces. Conceptually, the MVC framework is a general three-way factoring of interactive systems, with the Model representing the application, the View presenting the application's internal state to the user, and the Controller managing interaction with the user. However, its realization within Smalltalk results in a rigid, inflexible class hierarchy that only facilitates the creation of specific interfaces. Furthermore, the exact boundaries between the three components are neither enforced nor maintained. One of the major problems with the implementation is that the exact nature of the behavior of the Model (or application) is not specified. This leads to interactive systems whose user interface is tightly bound to the application, resulting in inflexible interfaces.

Implementing the MIMD UIDE made it clear that a general graphics/windowing platform is crucial for the development of flexible UIDEs. Unfortunately, SIMD UIDEs, such as Smalltalk, do not provide such a general graphics/windowing subsystem. This implementation also validated the claim that a clean separation of the user interface from the application component is necessary for the development of flexible UIDEs. The modifications that were made to generalize Smalltalk's SIMD UIDE represent an attempt at defining such a complete separation.

In conclusion, the experiences with implementing a prototype of the MIMD UIDE has strengthened the validity of the models and design that were developed for the MIMD UIDE. It has also shown that current object-oriented environments are not suitable for developing flexible UIDEs. It is therefore necessary to build an MIMD UIDE from scratch, on top of a flexible, device independent, general-purpose graphics/windowing subsystem that does not impose any biases on interaction styles, techniques and devices. It is important to note that standards for graphics/windowing systems do not currently exist. However, the current indication is that such general-purpose, object-oriented graphics/windowing systems will be marketed in the very near future.

6.2. Further Research In MIMD UIDEs

The major aim of further research in the immediate future will be to develop a complete MIMD UIDE. This development will be carried out on two platforms. The first platform is Smalltalk/V on the Macintosh, which has added the necessary modifications to Smalltalk's support for user interfaces to make it more suitable as a development environment. The second platform will be based on C++. Since C++ is classless, it presents a platform that facilitates an MIMD UIDE implementation that exactly follows the design. X windows will be used as the support for the underlying graphics/windowing system.

The prototype will then be converted into a fully operational, effective MIMD UIDE. Such an attempt will require further research into the following three areas: interactive data structures, interaction technologies, and the development of efficient user interfaces. Each of these areas is discussed below.

Interactive Data Structures

The effectiveness and efficiency of interactive data structures is largely dependent on the specification and maintenance of neighborhood relationships for both the internal and especially the external representations. Our design of the neighborhood relationships for the external representation used an ad-hoc block that (re)calculated every node's display geometry. Whenever a change occurred in the data structure, this brute-force recalculation was applied.

Furthermore, certain natural extensions of interactive data structures also require the definition and maintenance of dependencies. The following extensions would enhance the generality of the interactive data structure model.

[1] This research was concerned with developing user interfaces for interactive systems that comprise a single application. The current trend, however, is to develop sophisticated interactive environments that integrate multiple applications which are presented to the user as a single system. To accommodate such complex interactive systems, it is necessary to extend the model to include many applications implying multiple interactive data structures. The major problems will be in defining dependencies between these structures, and maintaining these dependencies whenever any data structure changes.

[2] The design of the data structure hierarchy included only the standard base data structures (lists, trees, sets, graphs) and their variants. However, sophisticated interactive applications and environments require more complex structures. One possible way to extend the interactive data structure model is to allow the base structures to be pasted (combined) together to form complex structures. Once again, the major issue to be resolved is the definition and maintenance of dependencies between the components of the complex data

structure.

There exist three models for specifying and maintaining dependencies that are worth considering: Smalltalk's dependencies, semantic attributes and constraints. Each of these models is discussed below.

Smalltalk's model for dependencies was presented in the previous Chapter. To reiterate, every object can maintain a list of objects that are dependent on it. Whenever an object changes its internal state, it can broadcast a (parameterized) change message to its dependents. These dependent objects react to the change through locally defined updating schemes. The major advantage of such a loose model is its simplicity for defining the dependencies. The major disadvantage is that the maintenance of these dependencies must be provided by the designer and is therefore carried out in an ad-hoc manner.

In the initial work for this research, context-free grammars were extended to 2-dimensions in order to develop a formalism for defining graphical applications [85]. Figure 6.1 shows an example of using an attributed graphical grammar to define the language of D-Charts [111]. The major problem with defining such graphical languages is the specification of the display geometry. This problem was solved by defining a rectangular region (width, height and entry point) representing the minimum area necessary to display each terminal and nonterminal. Each production in the grammar defines a set of equations that specify how the rectangular region of the left side nonterminal can be computed in terms of the rectangular areas associated with symbols on the right. As an example of these display equations, consider the expansion of a <D-Chart> nonterminal by applying production 1, as shown in Figure 6.2. The dimensions of the rectangular region enclosing the expanded <D-Chart> is given by

W' = max (width(ENTRY), width(<Body>), width(EXIT));
H' = height(ENTRY) + height(<Body>) + height(EXIT).

Thus, the geometric attributes for the display attributes for <D-Chart> are **synthesized** from the geometric attributes of symbols on the right side of production 1. However, not all geometric

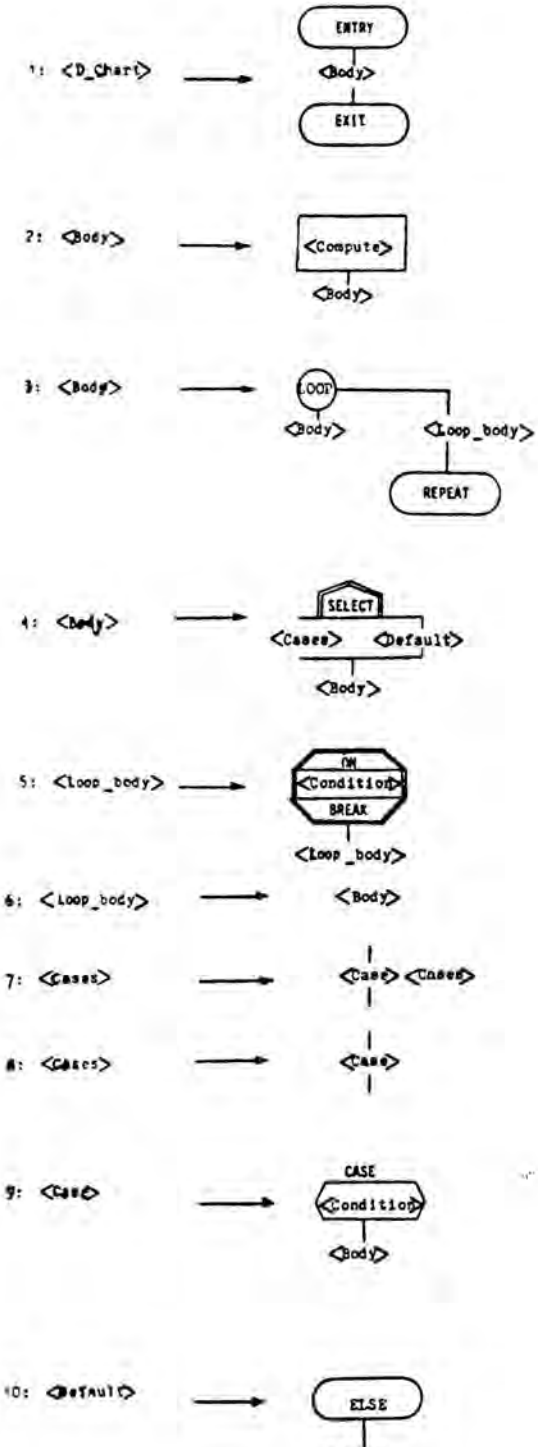
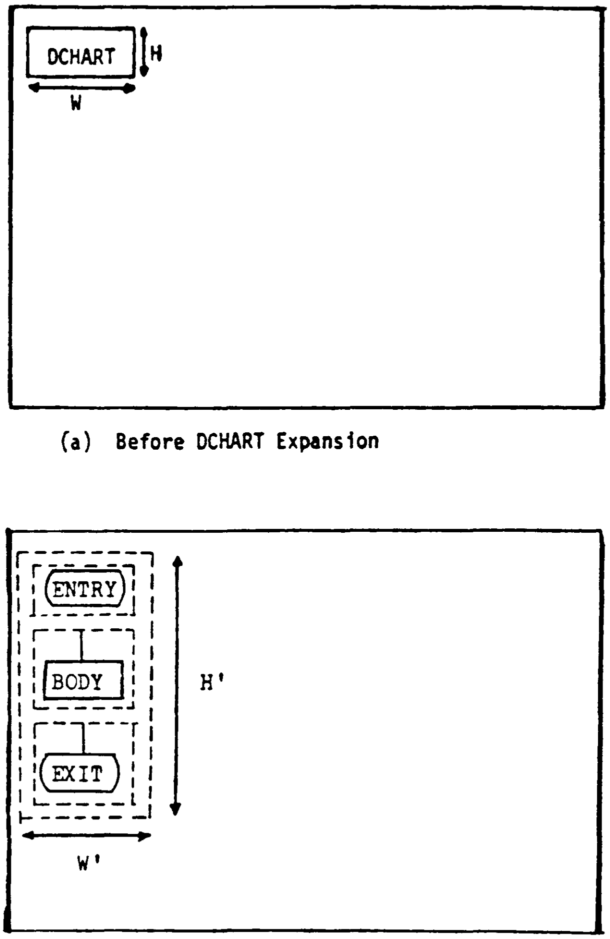


Figure 6.1: A Sample D-Chart Grammar



(a) Before DCHART Expansion

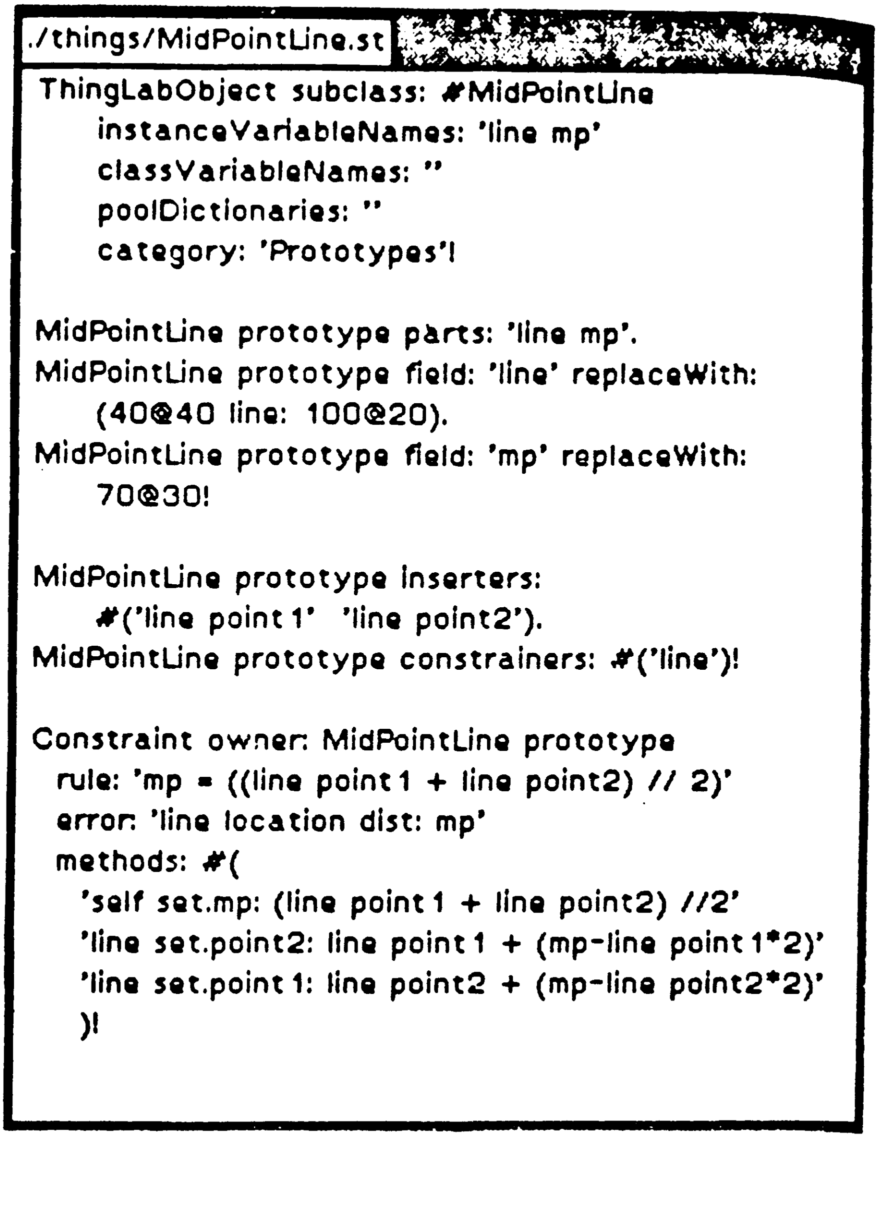
(b) After DCHART Expansion

Figure 6.2: A D-Chart Expansion

attributes can be synthesized. For example, the display of D-Chart conditional structures (given by productions 4, 7, 8, 9 and 10) must ensure that the vertical dimension of the <Default> be the same as the vertical dimension of <Cases>. To force this alignment, both <Cases> and <Default> inherit their height attribute from their common parent, <Body> in production 4. Thus, synthesized attributes pass information up the associated syntax tree while inherited attributes pass information down or across the tree. Reps [112] has devised incremental algorithms to ensure the consistency of these (semantic) attributes whenever a change is made to the tree. When applied to defining dependencies within interactive data structures, the semantic attributes model has the advantage of automatically maintaining the dependencies. The major disadvantage is that such semantic equations are difficult to specify.

Constraints specify relationships that must be maintained. A constraint that **A = B + C** could be used to determine A given B and C, but could also be used to determine B given A and C, and C given A and B. That is, constraints are multi-directional. A set of constraints are transformed into executable procedures that automatically maintain the specified relations. Figure 6.3 shows an example of a constraint MidPointLine which consists of a line and a point and the constraint that the point is always in the middle of the line. Constraints are versatile as is evident from their use in a wide variety of applications including geometric layout [113], physical simulation and algorithm animation [114], document formatting [115], design and analysis of mechanical devices and electrical circuits [116], jazz improvisations [117], and user interfaces [118]. The major advantages of using constraints to maintain relationships is that they are declarative in nature, and particularly suited to graphical techniques. Figure 6.4 shows a graphical definition of MidPointLine [119]. The major disadvantage of constraints is that the algorithms for maintaining them are complex, and for a large set of constraints extremely slow. However, ongoing research on constraints may remedy this disadvantage in the very near future.

Another important extension of interactive data structures is how they can be animated. Animation of data structures and algorithms are very useful in simulating interactive systems and



```
ThingLabObject subclass: #MidPointLine
     instanceVariableNames: 'line mp'
     classVariableNames: ''
     poolDictionaries: ''
     category: 'Prototypes'!

MidPointLine prototype parts: 'line mp'.
MidPointLine prototype field: 'line' replaceWith:
     (40@40 line: 100@20).
MidPointLine prototype field: 'mp' replaceWith:
     70@30!

MidPointLine prototype inserters:
     #('line point1'  'line point2').
MidPointLine prototype constrainers: #('line')!

Constraint owner: MidPointLine prototype
  rule: 'mp = ((line point1 + line point2) // 2)'
  error: 'line location dist: mp'
  methods: #(
    'self set.mp: (line point1 + line point2) //2'
    'line set.point2: line point1 + (mp-line point1*2)'
    'line set.point1: line point2 + (mp-line point2*2)'
    )!
```

Figure 6.3: Specification of MidPointLine

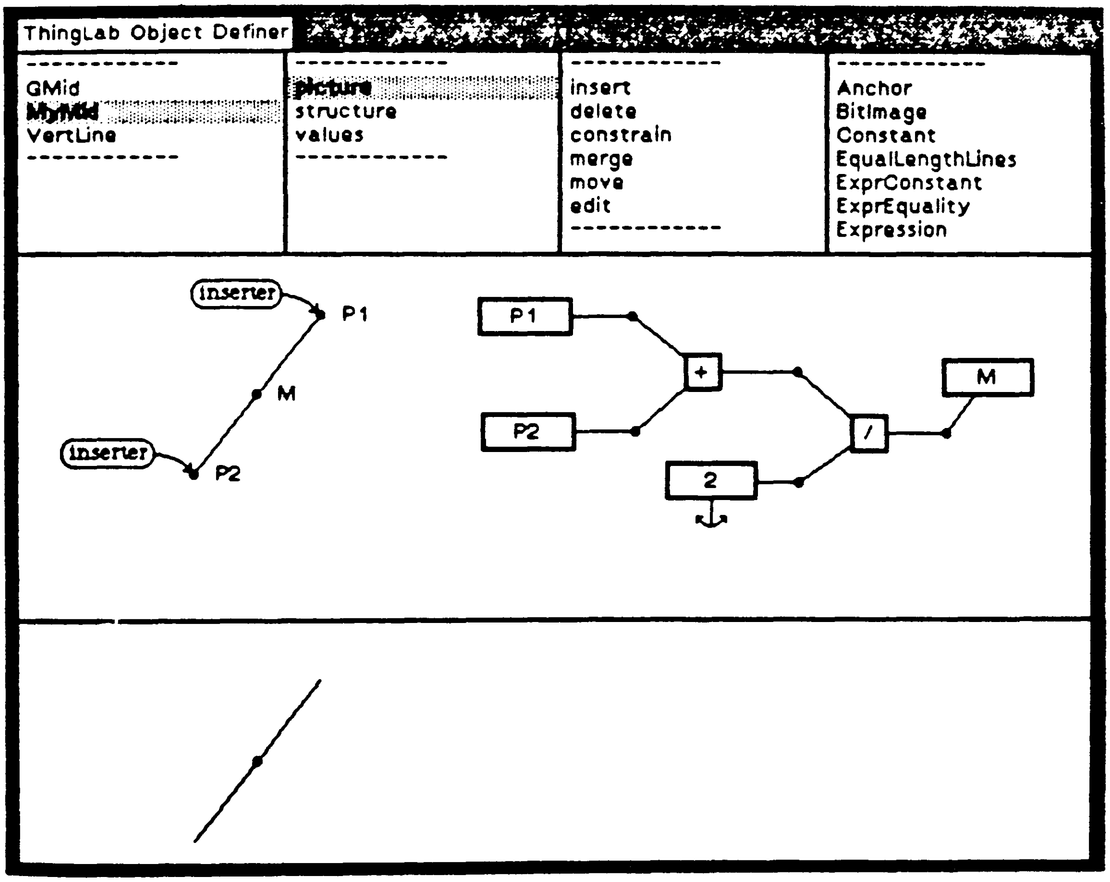
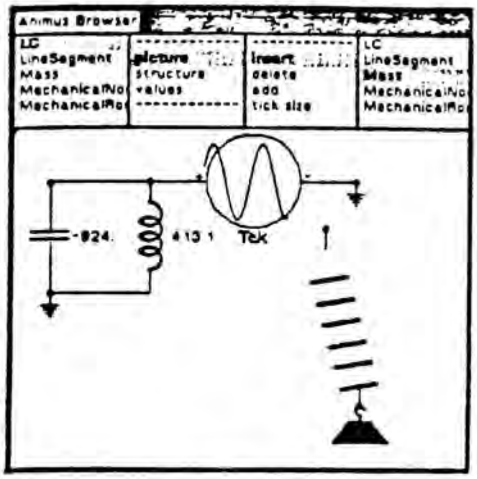


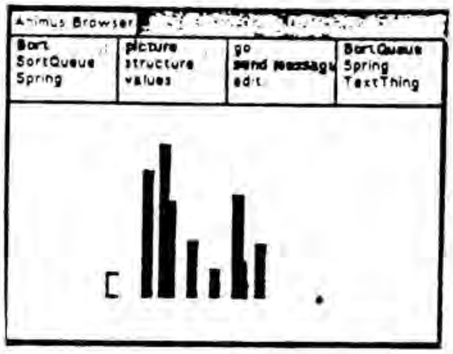
Figure 6.4: Graphical Specification of MidPointLine

in manipulating the runtime state of the system for debugging. Two approaches for animating interactive data structures are worth considering. The first is the use of temporal constraints [114] to specify the evolution of data structures and their external views by discrete time increments. Figure 6.5 shows three examples of animation using temporal constraints. The second approach concerns the maintenance and manipulation of **temporal data** within databases. A temporal data is a triplet, **(O, T, A)** where O is the identifying object, T is the time, and A the attribute value of O at time T. To capture the evolution of O through a period of time, it is necessary to define a time sequence for O as **(O, <T,A>*).** Such temporal data can be subjected to temporal queries such as "get values for O in the time range from ta to tb." A survey of over 70 articles devoted to research in temporal databases can be found in [120].

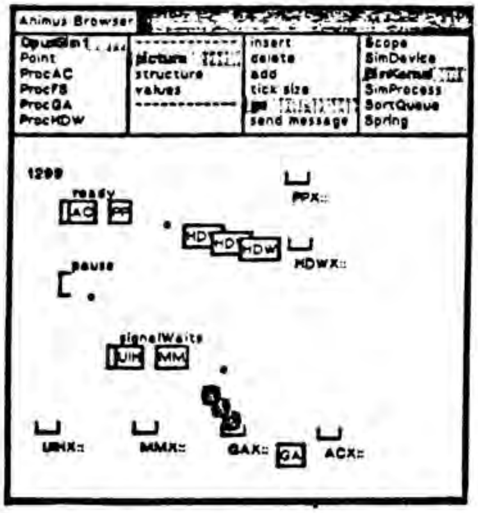
Note that both the temporal approaches presented above are mainly concerned with maintaining and manipulating the history of evolution for a particular structure. In the realm of animating interactive data structures, the main concern is with maintaining and manipulating the history of changes to the data structure. Thus, the initial state of the interactive data structure is copied to a **transcript** file. Every operation (along with its parameters) applied to the data structure is recorded in the transcript file. It would then be necessary to develop tools that can execute the transcript file, allowing the user to start and stop the animation at any point, reverse execute the transcript, etc. Such a facility would require the definition of methods that reverse (or undo) the effect of every method provided to manipulate the interactive data structure. Note that the discussion so far has concentrated on animating a single interactive data structure. To facilitate animation of the entire interactive system would require that all operations entered in a particular interactive data structure's transcript file be time stamped relative to the start of the session. This would provide the necessary information to synchronize the simultaneous animation of many interactive data structures.



Oscillating animations. The numbers indicate the charge on the capacitor, and the current through the inductor. The mass on the spring oscillates in two dimensions.



An animation of a selection sort caught in mid-swap.



An operating system animation. A Hardware process is proceeding from the ready list, to wait on its message exchange, and a message is being sent to the process GA (General Accounting.)

Figure 6.5: Animation Using Temporal Constraints

Interaction Technologies

There currently exist a large number of interaction styles, techniques and devices. To be effective, the MIMD UIDE must incorporate as many of these interaction technologies to provide the user a rich mix of possibilities for developing user interfaces. The modifications that will be necessary to incorporate these interaction technologies have already been outlined in Chapter 3. Initially, the MIMD UIDE will incorporate more interaction devices including joysticks, trackballs and voice. The next step is to complete the interaction tasks to include path, orient and quantify. Newer interaction technologies, such as three-dimensional graphical interaction, may require extensions to the model.

Efficiency

Finally, the MIMD UIDE must be concerned with efficiency. The main problem with graphical interaction is speed. This problem is more acute in the design of the MIMD UIDE due to the complete separation of the application from the user interface. This requires the user interface to communicate with the application for most of the information needed for interacting with the user. For example, since the user interface only maintains the display buffer of the external representation, it must communicate with the application to facilitate movement between objects in the display buffer. However, the more sophisticated workstations available today support graphical algorithms that are done entirely in hardware. These graphical architectures may be the solution to most of the speed problems.

6.3. Tool and Environment Generators

This dissertation has been mainly concerned with generating the user interface component of interactive systems. It defined a general model for the application component as a basis for developing a well-defined interface between the two components. To be an effective tool, it is necessary to expand and strengthen the application model to allow the automatic generation of applications. Such an application generator when combined with the user interface generator would result in a **interactive tool generator,** as depicted in Figure 6.6. Note that all the

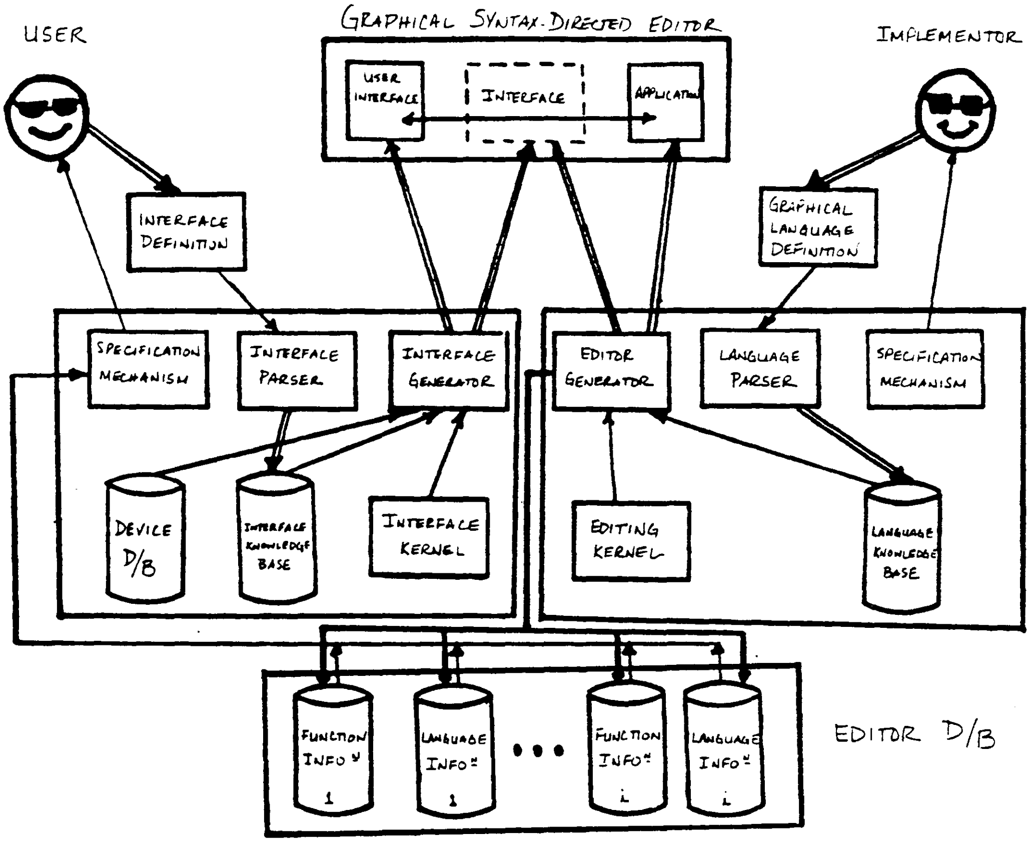


Figure 6.6: Interactive Tool Generator

applications that are generated are maintained in a database. This tool database would also maintain the set of user interfaces that were generated for each application. Such a system would provide an ideal basis for sharing and reusing at a very high level of granularity, that is, entire interactive tools!

The major issue to be resolved for developing such a tool generator is a 2-dimensional formalism that can be used to define the graphical language that describes the application. The previous section introduced the notion of attributed graphical grammars that extends conventional context-free grammars. The major problem of extending inherently 1-dimensional formalisms is that they result in poor specification mechanisms and more importantly increase the complexity of the design.

Arefi [86] uses the formalism of graph grammars to develop a general framework for the specification of graphical languages and their syntax-directed editors. The major contribution of her work is that the editing operations that manipulate the graphical language are defined within the specification of the language being edited. In other words, the specification of the graphical language includes the editing operations that can be performed on it. Figure 6.7 depicts the definition of a Petri-net editor using this formalism. Note that <del-place> and <del-trans> are delete operations applied to the places and transitions of the Petri-net, respectively.

The major problem with developing a practical formalism is the lack of a suitable 2-dimensional structure to represent the grammar that specifies the language. Note that syntax trees are perfect representations of context-free grammars as they provide a one-to-one mapping between the grammar and any string in the language. This strong representation is the main reason for the massive advancement in 1-dimensional language technology, which includes compiler generators, syntax-directed editors, editor generators, and environment generators. To be effective, formalisms for interactive graphical systems must also model a structure to represent the language that provides a one-to-one correspondence with the underlying grammar used to define it. It may be possible to use the interactive data structure model as the

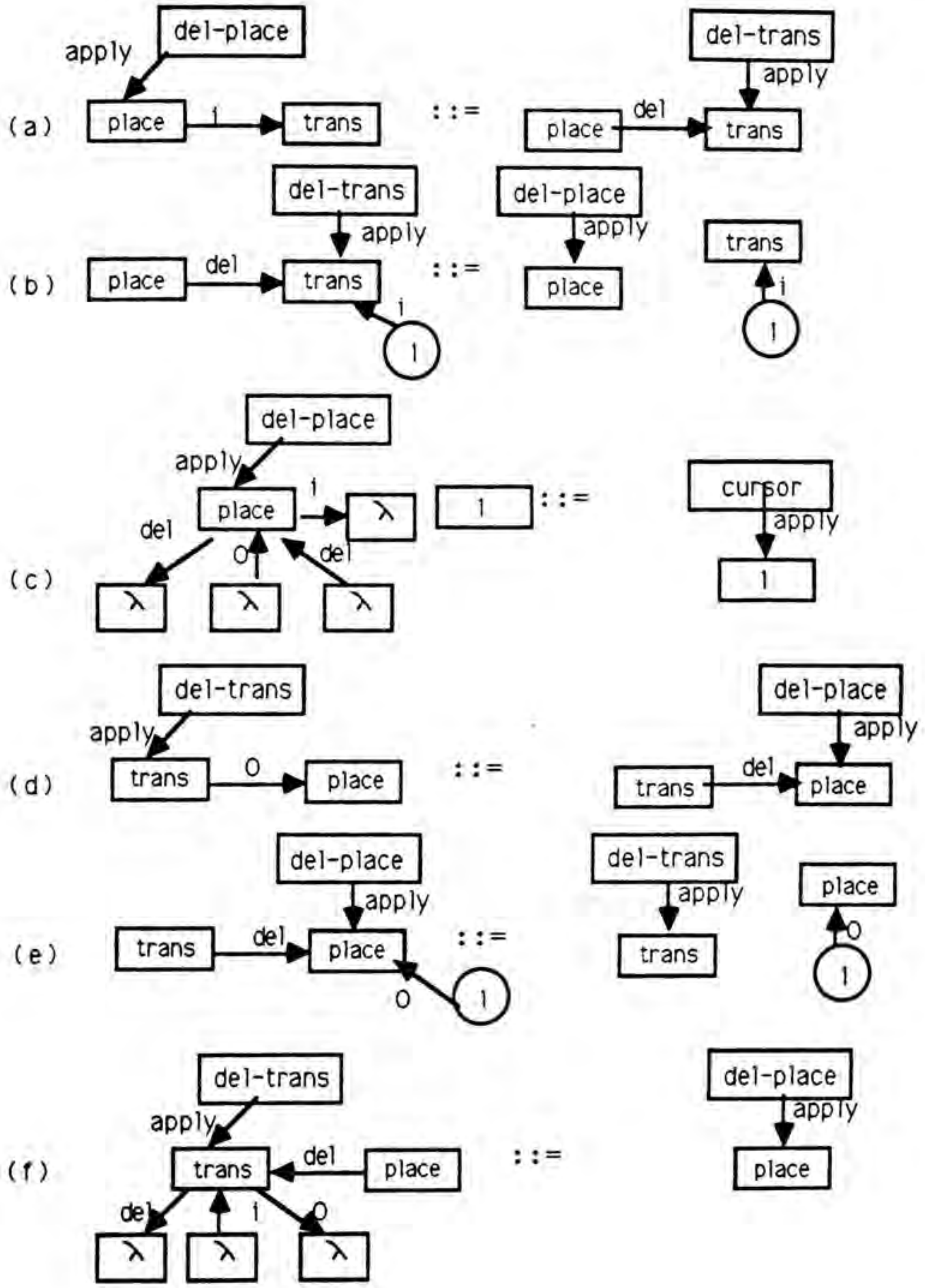


Figure 6.7: Graph Grammar Specification of Petri-Net Editor

basis for developing such a representation based on Arefi's formalism. It is important to note that formalisms developed for 2-dimensional graphical languages would provide an important theoretical foundation for the entire tool, including the user interface component.

Finally, the tool generator along with the theoretical formalisms would provide an ideal foundation for the development of an environment generator. An environment generator would be mainly concerned with specifying the combination of tools for the environment. For example, a typical programming environment consists of editors, linkers, and debuggers. The important aspects of generating an environment is the definition of interfaces and dependencies between these tools. Furthermore, an environment necessitates the definition of procedures to manage (e.g., version and change management) and control (e.g., access and version control) the environment as a whole.

When a truly evolvable and extendible environment generator has been developed, modern software engineering can ensure that software can progress at the same pace that hardware has in the last few decades. This research developed tools for the user interaction component of this environment generator. Hopefully, this work will motivate others to contribute to this new and important area of research.

APPENDICES

APPENDIX A

FORMAL DESIGN METHODOLOGIES FOR USER INTERFACE DEVELOPMENT

A first and crucial step towards the development of effective user interfaces must be the formulation of design methodologies which provide the basis for building interfaces. Central to a user-oriented design methodology is the definition of a user's model of the interactive system. The user's model provides the basis for designing the rest of the interactive system. This appendix reviews the research on developing effective design methodologies for the development of user interfaces.

What are the goals of an effective user interface design methodology, and what are its desirable features? Green [22] suggests three basic goals.

[1] **A formal notation** for describing the user interface should be provided by the design methodology. A formal notation is important for two major reasons. It ensures that a description of a user interface is interpreted in a uniform manner by the users of the notation; i.e. the personnel involved in the development of the user interface. More importantly, a formal notation serves as the basis for the implementation of the user interface from its description.

[2] The methodology should provide mechanisms for validating the man-machine interface design. Here we are interested in determining the correctness of the design before its implementation. The experiences of the research on Software Engineering has shown that it is hard, if not impossible, to show that a design is error free. The major concern is therefore the elimination of major bugs from the design before its implementation.

[3] The methodology should provide an effective framework for evaluating the quality of the user interface; that is, its effectiveness. Very little is known about the factors that affect the quality of the user interface. The methodology should at least provide a framework for measuring the human factors principles in the design. The following subsections present two impressive design methodologies for user interface development.

A.1 Green's Design Methodology For User Interfaces

The design methodology proposed by Green [22] is divided into two components; a formal description of the user's model and a formal specification of the user interface based on the user's model. The design methodology builds a separate user's model for every type of user that will be involved.

The description of the user's model comprises two components

[1] **a task model,** which formally describes the user's view of the problem in terms of atomic tasks. The task model is obtained through an informal task analysis, i.e. the decomposition of the problem into a number of tasks which are atomic from the user's point of view.

[2] **a control model,** which describes the actions that the user can perform.

Thus, the task model describes the tasks to be performed and the control model describes the commands that are provided to perform them; both from the user's viewpoint. Note that the control model must be consistent with the task model.

The notation employed for describing the task and control models is comprised of three components; object definitions, operator definitions, and invariants. Object definitions declare the properties (attributes) of the objects, similar to data structure definitions. Operator definitions describe the conditions for the application of an operator and the effects on the objects. Invariants describe the relations between the operators and objects in the model, and may vary greatly from user to user.

As an example, consider the design of a user interface based on the screen layout depicted in Figure A.1. The screen is divided into a menu area which depicts the geometrical shapes available to the user, and a work area, where the user creates pictures by arbitrary placement of the available geometric shapes. Three commands are supported: the **place** command allows the user to use a digitizing tablet to select a geometric shape from the menu and drag it into the work area; the **move** command which allows the user to move geometric shapes

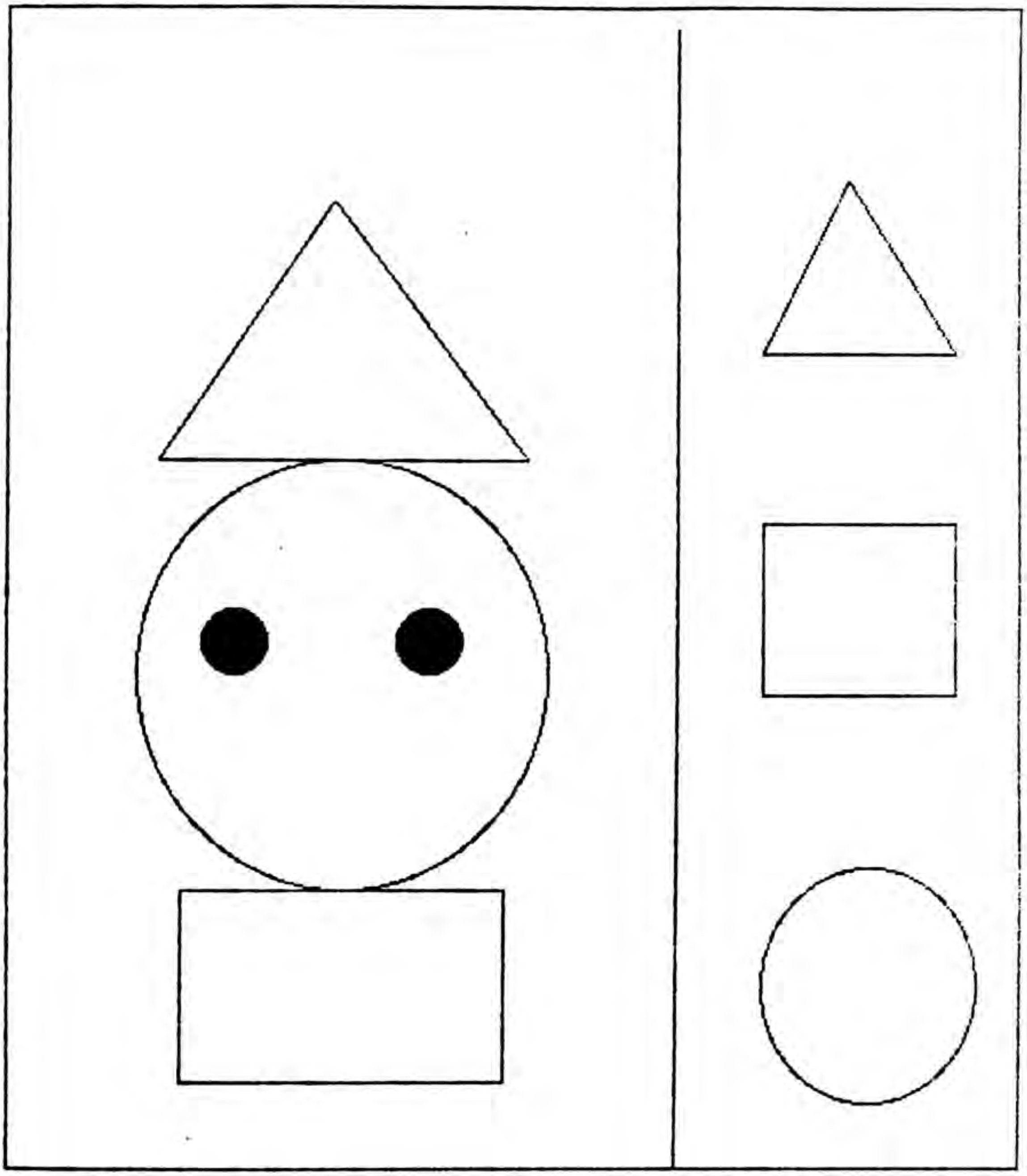


Figure A.1: A Simple Graphical Editor

around the work area; and the **remove** command which allows any geometric shape to be removed from the work area. The corresponding task and control models describing this user interface using the notation described above is presented in Figure A.2.

The next step in the design is the specification of the user interface from the high-level description provided by the control model. The specification language is based on state-transition diagrams and is influenced by SRI's SPECIAL language [23]. The specification comprises several state machines (called modules), where each machine is a collection of V and O functions. The V functions represent the machine state while the O functions describe transitions that change the machine's state. The specification of the example user interface is provided in Figure A.3.

The evaluation of the user interface specification to determine its correctness is described by the following three tests:

[1] the specification is **consistent** if each operator in the control model is implemented by the functions in the specification,

[2] all the invariants in the task and control models must hold for the specification, and

[3] testing the behavior of the specification under designer determined conditions.

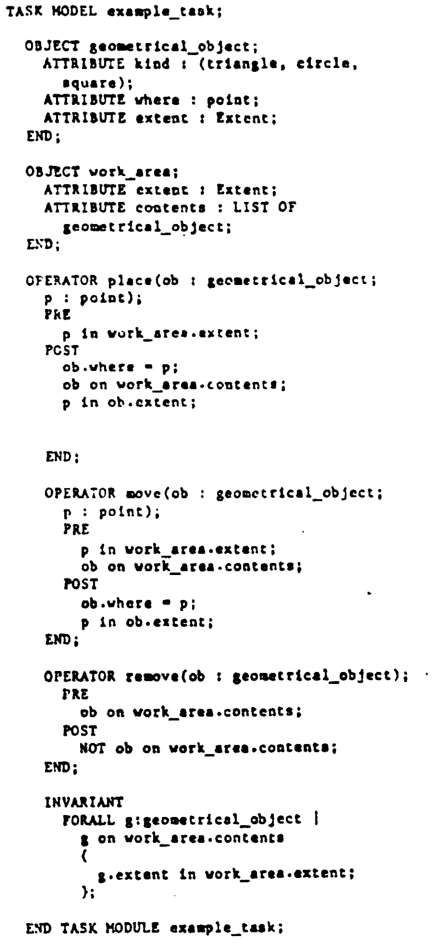
A.2 Moran's Command Language Grammar

Probably the most impressive design methodology developed to date is Moran's Command Language Grammar (CLG) [24]. The basic premise of Moran's proposal is that

to design the user interface of a system is to design the user's model.

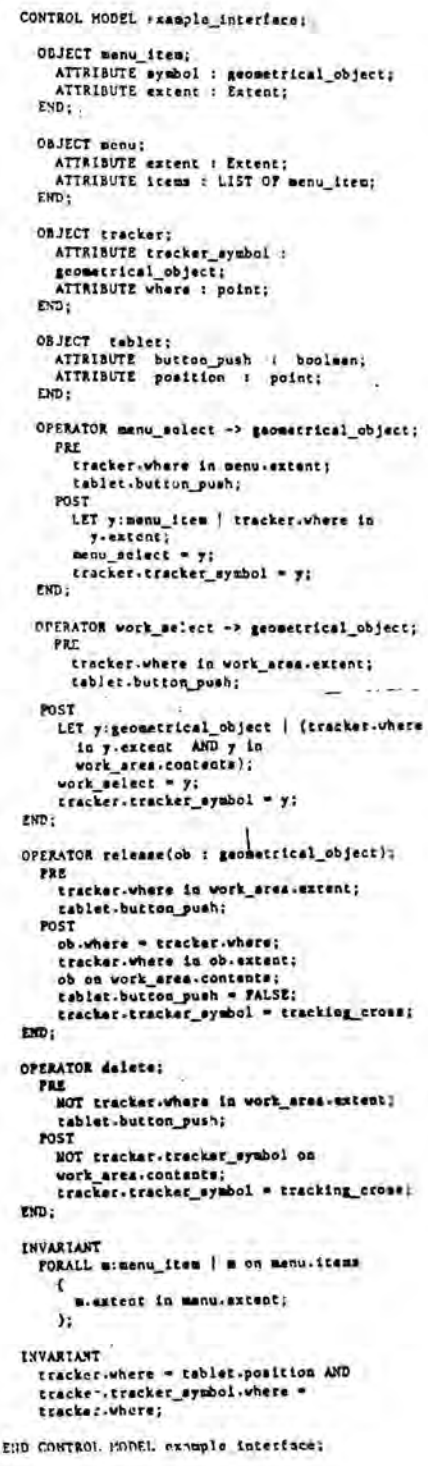
Thus a CLG representation of a system describes the user's conceptual model of the system.

The description of a user interface follows a top-down design methodology. The user's conceptual model is described, then a command language that implements the conceptual model is designed, and finally a display layout is designed to support the command language. The CLG representation is structured as a number of levels with the aim of separating the



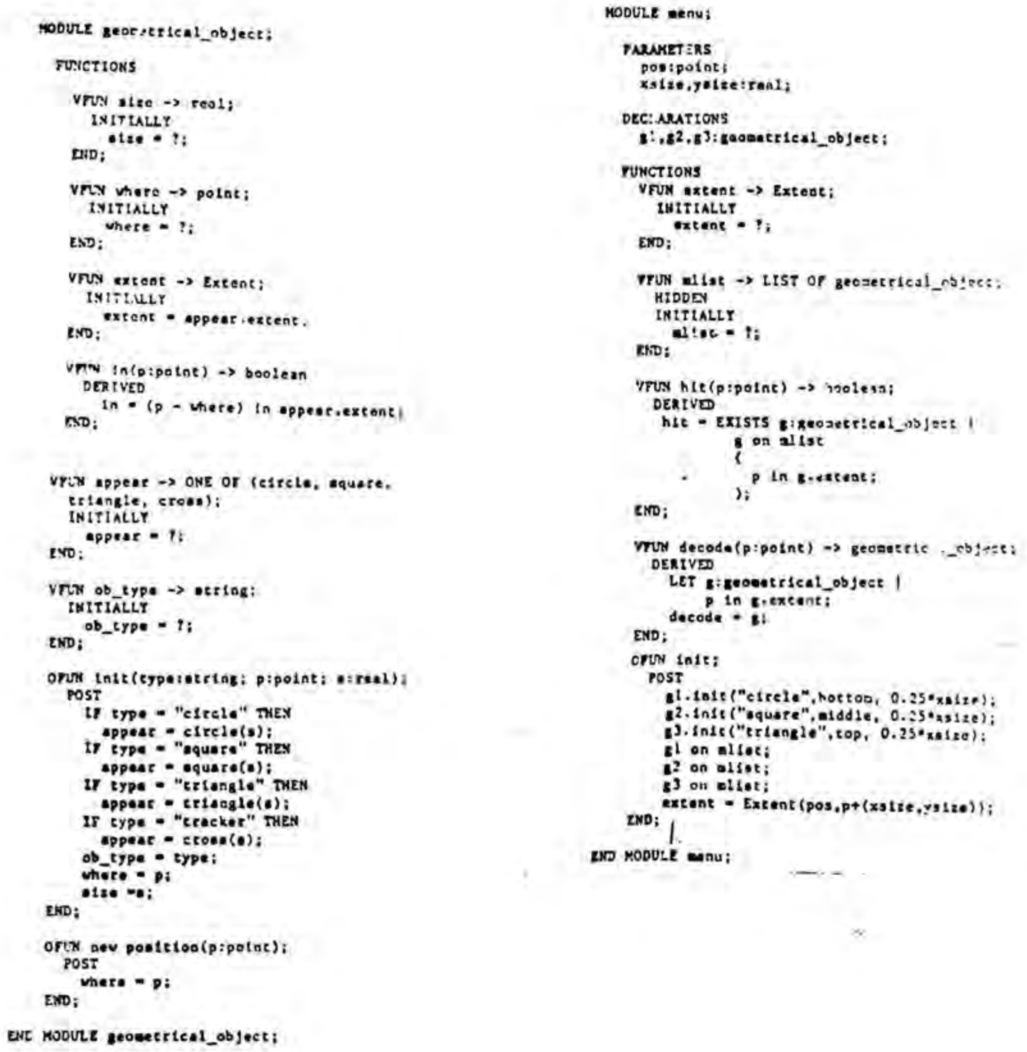
```
TASK MODEL example_task;

  OBJECT geometrical_object;
    ATTRIBUTE kind : (triangle, circle,
      square);
    ATTRIBUTE where : point;
    ATTRIBUTE extent : Extent;
  END;

  OBJECT work_area;
    ATTRIBUTE extent : Extent;
    ATTRIBUTE contents : LIST OF
      geometrical_object;
  END;

  OPERATOR place(ob : geometrical_object;
    p : point);
    PRE
      p in work_area.extent;
    POST
      ob.where = p;
      ob on work_area.contents;
      p in ob.extent;


    END;

    OPERATOR move(ob : geometrical_object;
      p : point);
      PRE
        p in work_area.extent;
        ob on work_area.contents;
      POST
        ob.where = p;
        p in ob.extent;
    END;

    OPERATOR remove(ob : geometrical_object);
      PRE
        ob on work_area.contents;
      POST
        NOT ob on work_area.contents;
    END;

    INVARIANT
      FORALL g:geometrical_object |
        g on work_area.contents
        (
          g.extent in work_area.extent;
        );

  END TASK MODULE example_task;
```

Figure A.2(a): Task Model for Example Interface



```
CONTROL MODEL example_interface;

  OBJECT menu_item;
    ATTRIBUTE symbol : geometrical_object;
    ATTRIBUTE extent : Extent;
  END;

  OBJECT menu;
    ATTRIBUTE extent : Extent;
    ATTRIBUTE items : LIST OF menu_item;
  END;

  OBJECT tracker;
    ATTRIBUTE tracker_symbol :
    geometrical_object;
    ATTRIBUTE where : point;
  END;

  OBJECT  tablet;
    ATTRIBUTE  button_push  :  boolean;
    ATTRIBUTE  position  :  point;
  END;

  OPERATOR menu_select -> geometrical_object;
    PRE
      tracker.where in menu.extent;
      tablet.button_push;
    POST
      LET y:menu_item | tracker.where in
        y.extent;
      menu_select = y;
      tracker.tracker_symbol = y;
  END;

  OPERATOR work_select -> geometrical_object;
    PRE
      tracker.where in work_area.extent;
      tablet.button_push;
   POST
     LET y:geometrical_object | (tracker.where
       in y.extent  AND y in
       work_area.contents);
     work_select = y;
     tracker.tracker_symbol = y;
END;

OPERATOR release(ob : geometrical_object);
  PRE
    tracker.where in work_area.extent;
    tablet.button_push;
  POST
    ob.where = tracker.where;
    tracker.where in ob.extent;
    ob on work_area.contents;
    tablet.button_push = FALSE;
    tracker.tracker_symbol = tracking_cross;
END;

OPERATOR delete;
  PRE
    NOT tracker.where in work_area.extent;
    tablet.button_push;
  POST
    NOT tracker.tracker_symbol on
    work_area.contents;
    tracker.tracker_symbol = tracking_cross;
END;

INVARIANT
  FORALL m:menu_item | m on menu.items
    {
      m.extent in menu.extent;
    };

INVARIANT
  tracker.where = tablet.position AND
  tracker.tracker_symbol.where =
  tracker.where;

END CONTROL MODEL example interface;
```

Figure A.2(b): Control Model for Example Interface



```
MODULE geometrical_object;

  FUNCTIONS

    VFUN size -> real;
      INITIALLY
        size = ?;
    END;

    VFUN where -> point;
      INITIALLY
        where = ?;
    END;

    VFUN extent -> Extent;
      INITIALLY
        extent = appear.extent.
    END;

    VFUN in(p:point) -> boolean
      DERIVED
        in = (p - where) in appear.extent;
    END;


   VFUN appear -> ONE OF (circle, square,
     triangle, cross);
     INITIALLY
       appear = ?;
   END;

   VFUN ob_type -> string;
     INITIALLY
       ob_type = ?;
   END;

   OFUN init(type:string; p:point; s:real);
     POST
       IF type = "circle" THEN
         appear = circle(s);
       IF type = "square" THEN
         appear = square(s);
       IF type = "triangle" THEN
         appear = triangle(s);
       IF type = "tracker" THEN
         appear = cross(s);
       ob_type = type;
       where = p;
       size =s;
   END;

   OFUN new position(p:point);
     POST
       where = p;
   END;

END MODULE geometrical_object;
```

```
MODULE menu;

  PARAMETERS
    pos:point;
    xsize,ysize:real;

  DECLARATIONS
    g1,g2,g3:geometrical_object;

  FUNCTIONS
    VFUN extent -> Extent;
      INITIALLY
        extent = ?;
    END;

    VFUN mlist -> LIST OF geometrical_object;
      HIDDEN
      INITIALLY
        mlist = ?;
    END;

    VFUN hit(p:point) -> boolean;
      DERIVED
       hit = EXISTS g:geometrical_object |
                g on mlist
                (
                  p in g.extent;
                );
    END;

    VFUN decode(p:point) -> geometrical_object;
      DERIVED
        LET g:geometrical_object |
            p in g.extent;
        decode = g;
    END;

   OFUN init;
     POST
       g1.init("circle",bottom, 0.25*xsize);
       g2.init("square",middle, 0.25*xsize);
       g3.init("triangle",top, 0.25*xsize);
       g1 on mlist;
       g2 on mlist;
       g3 on mlist;
       extent = Extent(pos,p+(xsize,ysize));
   END;

END MODULE menu;
```

Figure A.3: Interface Specification

```
  PARAMETERS
    pos:point;
    xsize,ysize:real;

  FUNCTIONS

    VFUN extent -> Extent;
      INITIALLY
        extent = ?;
    END;

    VFUN dlist -> LIST OF geometrical_object;
      HIDDEN
      INITIALLY
        dlist = ?;
    END;

    VFUN hit(p:point) -> boolean;
      DERIVED
        hit = EXISTS g:geometrical_object |
                  g on dlist
                  (
                    p in g.extent;
                  );
    END;

    VFUN decode(p:point) -> geometrical_object;
      DERIVED
        LET g:geometrical_object |
            p in g.extent;
        decode = g;
    END;

    OFUN include(g:geometrical_object);
      PRE
        g.extent in extent;
      POST
        g on dlist;
    END;

    OFUN remove(g:geometrical_object);
      PRE
        g on dlist;
      POST
        NOT g on dlist;
    END;

    OFUN init;
      POST
        extent = Extent(pos,pos+(xsize,ysize));
    END;

  END MODULE work_area;

FUNCTIONS

  VFUN button_state -> ONE OF (up,down);
    INITIALLY
      button_state = up;
  END;

  VFUN position -> point;
    INITIALLY
      position = ?;
  END;

  OFUN button_push;
    POST
     *button_state = down;
      tracker.select(position);
  END;

  OFUN button_release;
    POST
      button_state = up;
  END;

  OFUN change_position(p : point);
    POST
      position = p;
      tracker.move(position);
  END;

END MODULE tablet;
```

```
DECLARATIONS
  cross : geometrical_object;
  menu : menu;
  work_area : work_area;

FUNCTIONS

  VFUN cross_displayed -> boolean;
    HIDDEN
    INITIALLY
      cross_displayed = TRUE;
  END;

  VFUN tracker_symbol -> geometrical_object;
    INITIALLY
      tracker_symbol = cross;
  END;

  VFUN where -> point;
    INITIALLY
      where = ?;
  END;

  OFUN move(p : point);
    POST
      where = p;
      tracker_symbol.where = p;
  END;


  OFUN select(p : point);
    POST

     CASE p in menu.extent AND menu.hit(p)
     (
       LET g : geometrical_object |
         g = menu.decode(p);
       current_cross = g;
       tracker_symbol = FALSE;
     );
     CASE cross_displayed AND
       work_area.hit(p)
     (
       LET g : geometrical_object |
         g = work_area.decode(p);
       tracker_sycbol = g;
       cross_displayed = FALSE;
     );
     CASE p in work_area.extent AND
       NOT cross_displayed
     (
       work_area.include(tracker.symbol);
       tracker_symbol = cross;
       cross_displayed = TRUE;
     );
     CASE NOT cross_displayed AND
       NOT p in work_area.extent
     (
       work_area.remove(tracker_symbol);
       tracker_symbol = cross;
       cross_displayed = TRUE;
     );
 END;
```

Figure A.3 (contd): Interface Specification conceptual model of the system from its command language and to show the relationships between them. The six levels, each being a refinement of the previous level, are organized into three components which describe the CLG structure.

[1] **The Conceptual Component:** describes the organization of the system as abstract concepts. This component is comprised of the **task level** and the **semantic level.** The task level analyses the user's requirements to specify the structure of the tasks which describes the system from the user's viewpoint. The semantic level defines the methods for accomplishing task structures in terms of the objects and operations (that manipulate these objects) around which the system is built. The semantic level serves both the user and the system; it describes the conceptual entities and operations for the user, and correspondingly the data structures and procedures for the system.

[2] **The Communication Component:** describes the command language and the dialogue of the user interface. The **syntactic level** and **interaction level** make up the communication component. The syntactic level is a further refinement of the semantic level, describing the command language with which the user communicates to the system. The methods of the semantic level are described in terms of the commands developed at the syntactic level. The meaning of the commands are defined in terms of the operations described at the semantic level. The command languages are described in terms of basic syntactic elements: commands, arguments, contexts, and state variables. The interaction level specifies the syntactic level elements in terms of physical actions; i.e. primitive device techniques for input (e.g. keypresses) and display actions for output. The rules describing dialogue structure are also described at the interaction level.

[3] **The Physical Component:** describes the physical devices of the user interface. The **spatial layout level** describes the nature of the system's display at each point of interaction. It therefore is concerned with the arrangement of input/output devices and the graphics facilities for displays. The **device level** describes the physical properties of input/output

devices and the underlying graphics primitives. These levels make up the physical component.

The command language grammar is the specification mechanism used in describing the first four levels that make up the conceptual and communication components. The levels comprising the physical component has not been developed yet. The CLG notation, informally described in Figure A.4, is based on the concepts of frames [25, 26], schemata [27], semantic nets [28], and production rules [29].

To understand the complexity of defining the user interface using the CLG framework, Figures A.5 to A.17 depicts the descriptions of a user interface for a mail system, at the four levels. Each message of the mail system consists of a header and a body. The header specifies the sender, receiver, data and subject; while the body contains arbitrary text. Every message that is created is put in a message file for the specified user. The user interface described in the Figures is for a system, EG, that helps the user manage the message file. Figures A.5 and A.6 specifies the entities and tasks at the task level. Figures A.7, A.8 and A.9 describe respectively the conceptual entities, operations and methods of the semantic level. The corresponding entities, commands and methods of the syntactic level are developed in Figures A.10, A.11 and A.12, and A.13 respectively. Finally, the command interactions, rules for commands and arguments, and a description of methods specified at the interaction level are depicted in Figures A.14, A.15 and A.16, and A.17 respectively.

CLG is a symbolic notation for describing systems as conceptual structures. The basic objects of CLG notation are symbols and symbolic expressions. An expression, which is a structure of symbols, represents a concept by describing it, that is, by having its constituent symbols represent constituent aspects of the concept. An expression can be associated with a symbol as its definition The symbol can then stand for the expression and hence for its concept:

symbol → expression → concept

Notationally, symbols are arbitrary strings of characters, indicated by being in a sans serif typefont, like THIS. An expression is a list of symbols or subexpressions in parentheses. For example, Q = (X Y (W Z)) defines the symbol Q by an expression with three elements—two symbols, X and Y, and a subexpression, (W Z), with two symbols. The syntax of expressions is not arbitrary. There are three forms of expression in CLG.

The first form is the basic CLG expression, a hierarchic description. It describes a concept by declaring that it is an instance of another concept, plus some modifications. For example, CHAR = (AN ENTITY CODE = (A NUMBER)) defines CHAR to be an instance of ENTITY with a component called its CODE, which is a NUMBER. A descriptive expression thus consists of a prefix symbol, a type symbol, and a list of components. A component of an expression may be thought of as a symbol definition within the localized context of the expression. The symbol naming a component may also be used in other expressions. A component CODE of an expression CHAR can be unambiguously referred to by the expression (THE CODE OF CHAR). The prefix symbols A or AN indicate that an expression is a pattern describing a class concept. The prefix THE indicates that the expression describes a unique referent.

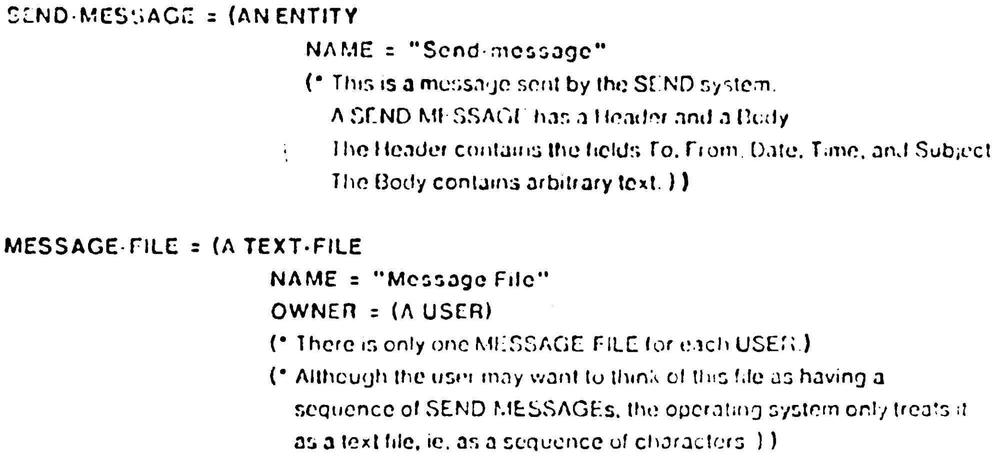
There are some minor variations to the syntax of this form of expression: The prefixes A or AN may be dropped. The symbol "=" is everywhere optional. A component may be unnamed. There is a special component, named OBJECT, that implicitly follows the type symbol. For example, (MOVE X FROM Y TO Z) has three components, the first of which, X, is the OBJECT of the MOVE.

The second form of expression is for representing collections of elements. It is indicated by a colon after the first symbol, which indicate the type of collection. For example: (SET: X Y Z) and (SEQ: X Y Z) represent, respectively, a set and a sequence of three elements. XYZ = (ONE-OF: X Y Z) defines XYZ to be the disjunction of the three items.

The third form of expression, (* ...), contains any English statement. It is usually used as a subexpression, allowing informal descriptions to be inserted anywhere within an expression structure.

An expression represents a concept, and its components represent the parts or aspects of that concept. Components can also represent the relations between expressions. For example, if T = (A TASK), P = (A PROCEDURE), and M = (A METHOD FOR T DO P), then M shows the relation between T and P. There are implicit relations between an expression and its component expressions. For example, the definition of P above should be (A PROCEDURE OF M), but the OF component does not need to be stated, since it is implicit. The other implicit relation is IN. For example, if S = (SET: X Y Z), then X, Y, and Z implicitly have the component IN S.

Figure A.4: CLG Notation



```
SEND-MESSAGE = (AN ENTITY
                        NAME = "Send-message"
                        (* This is a message sent by the SEND system.
                           A SEND MESSAGE has a Header and a Body
                           The Header contains the fields To, From, Date, Time, and Subject
                           The Body contains arbitrary text. ) )

MESSAGE-FILE = (A TEXT-FILE
                     NAME = "Message File"
                     OWNER = (A USER)
                     (* There is only one MESSAGE FILE for each USER.)
                     (* Although the user may want to think of this file as having a
                        sequence of SEND MESSAGEs, the operating system only treats it
                        as a text file, ie, as a sequence of characters ) )
```

Figure A.5: Task Level Description of Entities

```
GET-INFORMATION = (A TASK (* Get a piece of information from the last SEND MESSAGE
                             from a given person and delete the SEND-MESSAGE from
                             MESSAGE-FILE. ))

NEW-MAIL + THIN-OUT = (A TASK DO (SEQ: (NEW-MAIL)
                                        (THIN-OUT-MESSAGES) ))

NEW-MAIL = (A TASK (* Check for new SEND-MESSAGEs and, if any, read them.)
                   (* This is the most frequent task.)
                   DO (SEQ: (CHECK-FOR-NEW-MAIL)
                            (READ-NEW-MAIL) ) )

CHECK-FOR-NEW-MAIL = (A TASK (* Check to see if there are any new SEND MESSAGEs )
                               FAILURE = (* if no new SEND-MESSAGEs are found) )

READ-NEW-MAIL = (A TASK (* Read all new SEND-MESSAGEs, deleting those that are
                           not of further interest. ))

THIN-OUT-MESSAGES = (A TASK (* While in EG, thin out a long MESSAGE-FILE by deleting
                               the SEND-MESSAGEs that are no longer important. ))
```

Figure A.6: Task Level Description of Tasks

```
EG-SYSTEM = (A SYSTEM
                    NAME = "EG"
                    ENTITIES = (SET: MESSAGE SUMMARY
                                     MAILBOX DIRECTORY SCREEN)
                    OPERATIONS = (SET: SHOW DELETE))

MESSAGE = (AN ENTITY
                   REPRESENTS (A SEND-MESSAGE)
                   NAME = "Message"
                   AGE = (ONE-OF: OLD NEW)
                   (* A MESSAGE has a Header and a Body.
                      The Header contains the fields To, From, Date, Time, and Subject.
                      The Body contains arbitrary text. )
                   (* The AGE is a time-dependent mark; see MAILBOX. ))

SUMMARY = (AN ENTITY
                   OF (A MESSAGE)
                   NAME = "Summary"
                   (* This summarizes a MESSAGE in one line
                      by giving its AGE mark and its Header fields. ))

MAILBOX = (A LIST
                REPRESENTS (A MESSAGE-FILE)
                OWNER = (A USER)
                MEMBER = (A MESSAGE)
                NAME = "Mailbox"
                (* This contains all existing MESSAGEs. EG-SYSTEM puts all incoming
                   MESSAGEs at the end of the MAILBOX and marks them NEW.
                   All MESSAGEs from previous EG-SYSTEM sessions are marked OLD. ))

DIRECTORY = (A LIST
                  FOR (A MAILBOX)
                  MEMBER = (A SUMMARY)
                  NAME = "Directory" )

SCREEN = (AN ENTITY (* This is assumed to be large enough to display the DIRECTORY
                       or any MESSAGE. ))
```

Figure A.7: Semantic Level Description of Entities

```
DELETE = (A SYSTEM-OPERATION
              OBJECT = (A PARAMETER
                             VALUE = (A MESSAGE))
              (* The OBJECT is removed from MAILBOX and its SUMMARY is removed
                from DIRECTORY. ))

SHOW = (A SYSTEM OPERATION
              OBJECT = (A PARAMETER
                             VALUE = (AN ENTITY))
              IN (A PARAMETER
                        VALUE = (A PLACE ON SCREEN)
                        DEFAULT-VALUE = UNKNOWN)
              (* The OBJECT is shown to the USER at some place on the SCREEN.
                The OBJECT may be a MESSAGE, a SUMMARY, or the DIRECTORY.))

LOOK = (A USER-OPERATION
              IN (A PARAMETER
                        VALUE = (A PLACE ON SCREEN)
                        DEFAULT-VALUE = UNKNOWN)
              AT (A PARAMETER
                        VALUE = (AN ENTITY))
              FOR (A PARAMETER
                         VALUE = (PATTERN ENTITY))
              RESULT = (AN ENTITY (* that satisfies the FOR pattern))
              FAILURE = (* If nothing can be found that satisfies the FOR pattern.)
              (* The USER looks IN some place AT some entity FOR something,
                which is another entity. ))

READ = (A USER-OPERATION
              OBJECT = (A PARAMETER
                             VALUE = (AN ENTITY))
              IN (A PARAMETER
                        VALUE = (A PLACE ON SCREEN)
                        DEFAULT-VALUE = UNKNOWN)
```

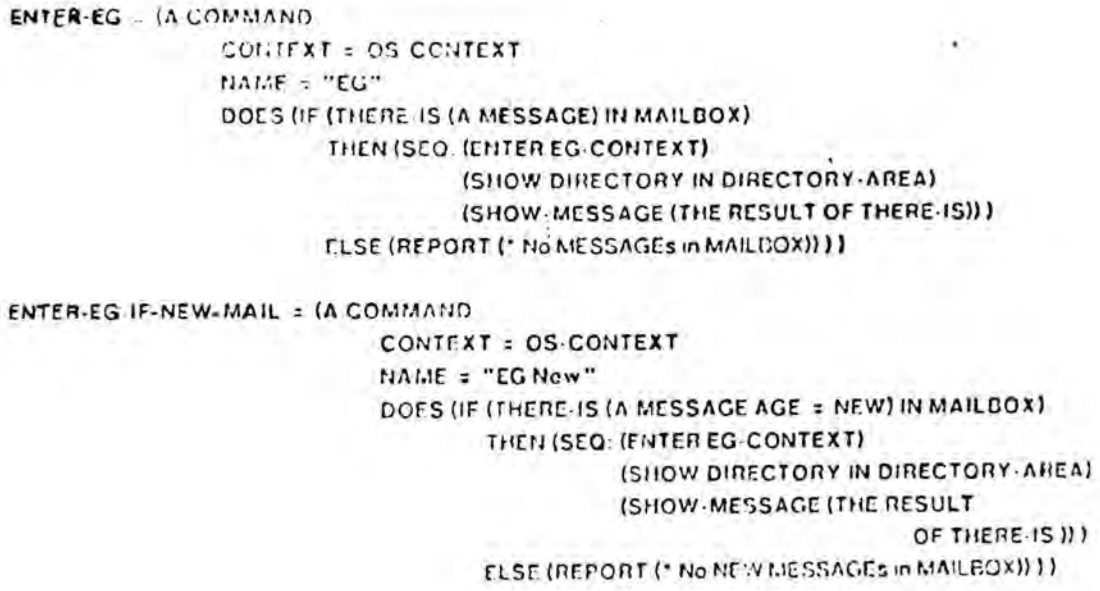
Figure A.8: Semantic Level Description of Operations

```
SEM-M1 = (A SEMANTIC-METHOD
                  FOR GET-INFORMATION
                  DO (SEQ: (START EG-SYSTEM)
                           (SHOW DIRECTORY)
                           (LOOK AT DIRECTORY FOR (A MESSAGE))
                           (SHOW (THE RESULT OF LOOK))
                           (READ (THE RESULT OF LOOK))
                           (DELETE (THE RESULT OF LOOK))
                           (STOP EG-SYSTEM)))

SEM-M2 = (A SEMANTIC-METHOD
                  FOR CHECK-FOR-NEW-MAIL
                  DO (SEQ: (START EG-SYSTEM)
                           (SHOW DIRECTORY)
                           (LOOK AT DIRECTORY FOR (A MESSAGE AGE = NEW))))

SEM-M3 = (A SEMANTIC-METHOD
                  FOR READ-NEW-MAIL
                  DO (REPEAT
                        BINDING m TO (EACH MESSAGE AGE = NEW)
                        DOING (SEQ: (SHOW m)
                                    (READ m)
                                    (OPT (DELETE m)))))

SEM-M4a = (A SEMANTIC-METHOD
                  FOR THIN-OUT-MAILBOX
                  DO (REPEAT
                        BINDING m TO (EACH MESSAGE)
                        DOING (SEQ: (SHOW (SUMMARY OF m))
                                    (READ (SUMMARY OF m))
                                    (OPT (SEQ: (SHOW m)
                                               (READ m)))
                                    (OPT (DELETE m)))))

SEM-M4b = (A SEMANTIC-METHOD
                  FOR THIN-OUT-MAILBOX
                  DO (REPEAT
                        UNTIL FAILURE
                        DOING (SEQ: (SHOW DIRECTORY)
                                    (LOOK AT DIRECTORY FOR (A MESSAGE))
                                    (OPT (SEQ: (SHOW (THE RESULT OF LOOK))
                                               (READ (THE RESULT OF LOOK))))
                                    (OPT (DELETE (THE RESULT OF LOOK))))))
```

Figure A.9: Semantic Level Description of Methods

```
EG-CONTEXT = (A COMMAND-CONTEXT
                 STATE-VARIABLES = (SET: CURRENT-MESSAGE)
                 DESCRIPTORS = (SET: MESSAGE-NO)
                 DISPLAY-AREAS = (SET: DIRECTORY-AREA MESSAGE-AREA
                                     COMMAND-AREA)
                 COMMANDS = (SET: SHOW-MESSAGE SHOW-NEXT-MESSAGE
                                  DELETE-CURRENT-MESSAGE QUIT-EG)
                 ENTRY-COMMANDS = (SET: ENTER-EG ENTER-EG-IF-NEW-MAIL))

CURRENT-MESSAGE = (A STATE-VARIABLE
                      CONTEXT = EG-CONTEXT
                      VALUE = (A MESSAGE)
                      NAME = "Current Message" )

MESSAGE-NO = (A DESCRIPTOR
                 NAME = "Message Number"
                 FORM = (AN INTEGER)
                 VALUE = (A MESSAGE)
                 DEFAULT-VALUE = (THE CURRENT-MESSAGE)
                 (* The MESSAGE-NO for each MESSAGE is its sequential position in
                   MAILBOX upon entering EG-CONTEXT  MESSAGE-NOs do not
                   change when MESSAGEs are DELETEd from MAILBOX. )
                 DOES (CASE: (IF (* the FORM of MESSAGE-NO less than 1 or greater
                               than the number of MESSAGEs in MAILBOX )
                             THEN (REPORT (* MESSAGE-NO out of bounds)
                                         IN COMMAND-AREA ))
                           (IF (* the FORM of MESSAGE-NO corresponds to a deleted MESSAGE)
                             THEN (REPORT (* deleted MESSAGE) IN COMMAND-AREA))
                           (RETURN (* the MESSAGE corresponding to the FORM of
                                    MESSAGE-NO ))))

EG-DISPLAY-AREA = (A DISPLAY-AREA ON SCREEN CONTEXT = EG-CONTEXT)

DIRECTORY-AREA = (AN EG-DISPLAY-AREA
                     NAME = "Directory Window" )

MESSAGE-AREA = (AN EG-DISPLAY-AREA
                   NAME = "Message Window" )

COMMAND-AREA = (AN EG-DISPLAY-AREA
                   NAME = "Command Window" )
```

Figure A.10: Syntactic Level Description of Entities



```
ENTER-EG = (A COMMAND
              CONTEXT = OS CONTEXT
              NAME = "EG"
              DOES (IF (THERE IS (A MESSAGE) IN MAILBOX)
                       THEN (SEQ. (ENTER EG-CONTEXT)
                                  (SHOW DIRECTORY IN DIRECTORY-AREA)
                                  (SHOW-MESSAGE (THE RESULT OF THERE-IS)) )
                       ELSE (REPORT (" No MESSAGEs in MAILBOX)) ) )

ENTER-EG IF-NEW-MAIL = (A COMMAND
                           CONTEXT = OS-CONTEXT
                           NAME = "EG New"
                           DOES (IF (THERE-IS (A MESSAGE AGE = NEW) IN MAILBOX)
                                    THEN (SEQ: (ENTER EG-CONTEXT)
                                               (SHOW DIRECTORY IN DIRECTORY-AREA)
                                               (SHOW-MESSAGE (THE RESULT
                                                                 OF THERE-IS )) )
                                    ELSE (REPORT (" No NEW MESSAGEs in MAILBOX)) ) )
```

Figure A.11: Syntactic Level Description of Commands to Enter Context

```
EG-COMMAND = (A COMMAND
                 CONTEXT = EG-CONTEXT)


SHOW-MESSAGE = (AN EG-COMMAND
                    NAME = "Message"
                    OBJECT = (AN ARGUMENT
                                  FORM = (A MESSAGE-NO))
                    DOES (SET: (SHOW (SUMMARY OF (THE OBJECT))
                                   IN DIRECTORY-AREA)
                             (SHOW (MESSAGE-NO OF (THE OBJECT))
                                   IN DIRECTORY-AREA)
                             (SHOW (THE OBJECT) IN MESSAGE-AREA))
                    SIDE-EFFECT = (BIND CURRENT-MESSAGE TO (THE OBJECT))
                    (* Displays the SUMMARY by highlighting the SUMMARY of OBJECT
                     in the DIRECTORY in DIRECTORY-AREA. ))

DELETE-CURRENT-MESSAGE = (AN EG-COMMAND
                             NAME = "Delete"
                             DOES (SEQ: (DELETE (THE CURRENT-MESSAGE))
                                      (IF (THERE-IS (A MESSAGE) IN MAILBOX)
                                        THEN (SHOW-NEXT-MESSAGE)
                                        ELSE (SEQ: (REPORT (* No more MESSAGEs
                                                                in MAILBOX )
                                                               IN COMMAND-AREA )
                                                        (EXIT FROM EG-CONTEXT)))))

SHOW-NEXT-MESSAGE = (AN EG-COMMAND
                         NAME = "Next Message"
                         DOES (IF (THERE-IS (A MESSAGE) IN MAILBOX
                                              AFTER (THE CURRENT-MESSAGE))
                                THEN (SHOW-MESSAGE (THE RESULT OF THERE-IS))
                                ELSE (SEQ: (REPORT (* End of MAILBOX)
                                                       IN COMMAND-AREA )
                                               (SHOW-MESSAGE 1))))


QUIT-EG = (AN EG-COMMAND
              NAME = "Quit"
              DOES (EXIT FROM EG-CONTEXT))
```

Figure A.12: Syntactic Level Description of Context Commands

```
SYN-M1 = (A SYNTACTIC-METHOD
             FOR GET-INFORMATION
             DO (SEQ: (ENTER-EG)
                    (LOOK IN DIRECTORY-AREA AT DIRECTORY
                         FOR (A MESSAGE-NO))
                    (SHOW-MESSAGE (THE RESULT OF LOOK))
                    (READ (THE CURRENT-MESSAGE) IN MESSAGE-AREA)
                    (DELETE-CURRENT-MESSAGE)
                    (QUIT-EG)))

SYN-M2 = (A SYNTACTIC-METHOD
             FOR CHECK-FOR-NEW-MAIL
             DO (ENTER-EG-IF-NEW-MAIL))

SYN-M3 = (A SYNTACTIC-METHOD
             FOR READ-NEW-MAIL
             DO (REPEAT UNTIL (* End of MAILBOX)
                  DOING (SEQ: (READ (THE CURRENT-MESSAGE)
                                    IN MESSAGE-AREA )
                              (CHOICE: (SHOW-NEXT-MESSAGE)
                                       (DELETE-CURRENT-MESSAGE)))))

SYN-M4a = (A SYNTACTIC-METHOD
             FOR THIN-OUT-MAILBOX
             DO (SEQ: (IF (NOT-EQUAL (MESSAGE-NO OF (THE CURRENT-MESSAGE))
                                     TO 1)
                          THEN (SHOW-MESSAGE 1))
                       (REPEAT UNTIL (* End of MAILBOX)
                          DOING (SEQ: (READ (SUMMARY
                                                 OF (THE CURRENT-MESSAGE))
                                              IN DIRECTORY-AREA )
                                       (OPT (READ (THE CURRENT-MESSAGE)
                                                  IN MESSAGE-AREA ))
                                       (OPT (DELETE-CURRENT-MESSAGE))
                                       (SHOW-NEXT-MESSAGE)))))

SYN-M4b = (A SYNTACTIC-METHOD
             FOR THIN-OUT-MAILBOX
             DO (REPEAT UNTIL FAILURE
                  DOING (SEQ: (LOOK IN DIRECTORY-AREA AT DIRECTORY
                                    FOR (A MESSAGE-NO))
                              (SHOW-MESSAGE (THE RESULT OF LOOK))
                              (OPT (READ (THE CURRENT-MESSAGE)
                                         IN MESSAGE-AREA ))
                              (OPT (DELETE-CURRENT-MESSAGE)))))
```

Figure A.13: Syntactic Level Description of Methods

Non-terminal Constituents:

S	=	"the Specification"
B	=	"the Body of"
D	=	"the Designation of/in"
F	=	"the Form of/in"
T	=	"the Termination of"
I	=	"the Interpretation of"

Terminal Constituents:

W	=	"When is"
P	=	"the Prompt for"
A	=	"the Action for/in"
R	=	"the Response to"

```
 1  (W.S      OF SHOW-MESSAGE)                → (ANYTIME IN EG-CONTEXT)
 2  (P.S      OF SHOW-MESSAGE)                → (DISPLAY "Command: ")
 3  (W.D.B.S  OF SHOW-MESSAGE)                → (FIRST IN (B.S OF SHOW-MESSAGE))
 4  (A.D.B.S  OF SHOW-MESSAGE)                → (KEY: "M")
 5  (R.D.B.S  OF SHOW-MESSAGE)                → (DISPLAY "Message ")
 6  (W.S      OF (OBJECT OF SHOW-MESSAGE))
                                              → (AFTER (D.S OF SHOW-MESSAGE))
 7  (P.S      OF (A MESSAGE-NO))              → (DISPLAY " # ")
 8  (A.B.S    OF (AN INTEGER))                → (KEY: (THE INTEGER))
 9  (A.T.S    OF (A MESSAGE-NO))              → (KEY: RETURN)
10  (R.S      OF (A MESSAGE-NO))              → (DISPLAY (THE MESSAGE-NO))
11  (W.I.S    OF (OBJECT OF SHOW-MESSAGE))
                                              → (BEFORE (I OF SHOW-MESSAGE))
12  (W.I.S    OF SHOW-MESSAGE)                → (AFTER (S OF SHOW-MESSAGE))
```

Figure A.14: Interaction Level Description of Show-Message

```
CR1 = (RULE FOR (W.S OF (AN EG-COMMAND))
                 → (ANYTIME IN EG-CONTEXT))

CR2 = (RULE FOR (P.S OF (AN EG-COMMAND))
                 → (DISPLAY "Command:" IN COMMAND-AREA))

CR3 = (RULE FOR (W.D.B.S OF (AN EG-COMMAND))
                 → (FIRST IN (B.S OF (THE COMMAND))))

CR4 = (RULE FOR (A.D.B.S OF (AN EG-COMMAND))
                 → (KEY: (* First letter of the COMMAND NAME)))

CR5 = (RULE FOR (R.D.B.S OF (AN EG-COMMAND))
                 → (DISPLAY (* Full NAME of the COMMAND) IN COMMAND-AREA))

CR6 = (RULE FOR (W.I.S OF (AN EG-COMMAND))
                 → (AFTER (S OF (THE COMMAND))))
```

Figure A.15: Interaction Level Description of Command Rules

```
AR1 = (RULE FOR (W.S OF (AN ARGUMENT
                           OF (AN EG-COMMAND)
                           VALUE = (A MESSAGE) ))
          → (AFTER (D.B.S OF (THE COMMAND))) )

AR2 = (RULE FOR (P.S OF (AN ARGUMENT
                          OF (AN EG-COMMAND)
                          FORM = (A MESSAGE-NO) ))
          → (DISPLAY "Ø" IN COMMAND-AREA) )

AR3 = (RULE FOR (A.B.S OF (AN INTEGER))
          → (KEY: (THE INTEGER)) )

AR4 = (RULE FOR (A.T.S OF (A MESSAGE-NO))
          → (KEY: RETURN) )

AR5 = (RULE FOR (R.S OF (A MESSAGE-NO))
          → (DISPLAY (IF (DEFAULTED (A.S OF (THE MESSAGE-NO)))
                        THEN (MESSAGE-NO OF (THE CURRENT-MESSAGE))
                        ELSE (THE MESSAGE-NO) )
                     IN COMMAND-AREA ) )

AR6 = (RULE FOR (W.I.S OF (AN ARGUMENT OF (AN EG-COMMAND)))
          → (BEFORE (I.S OF (THE COMMAND))) )
```

Figure A.16: Interaction Level Description of Argument Rules

```
IA-M1  = (AN INTERACTION-METHOD
              FOR GET-INFORMATION
              DO (SEQ: (KEY: "EG" RETURN)
                     (LOOK IN DIRECTORY-AREA AT DIRECTORY
                          FOR (A MESSAGE-NO))
                     (KEY: "M" (A MESSAGE-NO) RETURN)
                     (READ (THE CURRENT-MESSAGE) IN MESSAGE-AREA)
                     (KEY: "D" "Q")))

IA-M2  = (AN INTERACTION-METHOD
              FOR CHECK-FOR-NEW-MAIL
              DO (KEY: "EG/N" RETURN))

IA-M3  = (AN INTERACTION-METHOD
              FOR READ-NEW-MAIL
              DO (REPEAT UNTIL (* End of MAILBOX)
                   DOING (SEQ: (READ (THE CURRENT-MESSAGE)
                                    IN MESSAGE-AREA)
                              (CHOICE (KEY: "N")
                                      (KEY: "D")))))

IA-M4a = (AN INTERACTION-METHOD
              FOR THIN-OUT-MAILBOX
              DO (SEQ: (IF (NOT-EQUAL (MESSAGE-NO OF (THE CURRENT-MESSAGE))
                                    TO 1)
                         THEN (KEY: "M" "1" RETURN))
                       (REPEAT UNTIL (* End of MAILBOX)
                         DOING (SEQ: (READ (SUMMARY
                                              OF (THE CURRENT-MESSAGE))
                                           IN DIRECTORY-AREA)
                                    (OPT (READ (THE CURRENT-MESSAGE)
                                              IN MESSAGE-AREA))
                                    (OPT (KEY: "D"))
                                    (KEY: "N")))))

IA-M4b = (AN INTERACTION-METHOD
              FOR THIN-OUT-MAILBOX
              DO (REPEAT UNTIL FAILURE
                   DOING (SEQ: (LOOK IN DIRECTORY-AREA AT DIRECTORY
                                    FOR (A MESSAGE-NO))
                              (KEY: "M" (A MESSAGE-NO) RETURN)
                              (OPT (READ (THE CURRENT-MESSAGE)
                                      IN MESSAGE-AREA))
                              (OPT (KEY: "D")))))
```

Figure A.17: Interaction Level Description of Methods

APPENDIX B

AXIOMATIC MODELS FOR USER INTERFACE SPECIFICATION

Most axiomatic specification techniques are based on algebraic axioms [36]. The algebraic specification approach describes a class of objects by defining the set of functions which operate on the class. The syntax of each function is described by its name, domain and range. The semantics of the function is defined by a set of algebraic axioms which must be finite, and can only contain operations of the given type or other previously defined types, global variables and conditional tests. The functions are grouped into two categories: **generator** functions and **inquiry** functions. Generator functions have ranges of the given object class, while inquiry function have ranges outside the object class. The advantages of using an algebraic specification technique are described below.

[a] It provides a simple notation which is easy to comprehend; promoting clarity and conciseness.

[b] It is easy to construct a specification which is complete and usually consistent. A specification is complete if every inquiry operation applied to every object of a given type is defined by the axioms. This requires that axioms be defined for each application of an inquiry function to a generator function. Consistency implies that each inquiry operation associates at most one value with each object of the type.

[c] The specification is extensible, in the sense that minor changes in concepts result in minor changes in the specification.

[d] The specification allows for formal reasoning about the design of the user interface before its implementation. Such reasoning, usually through theorems, can prove properties of the specification.

[e] Most importantly, the specification allows the user interface to be implemented directly, and exactly matches the behavior of the specified system.

Chi [121] compares and evaluates four axiomatic approaches to the formal specification of user interfaces. All the approaches are used to define the user interface of a commercial line editor, which has an 80 character edit buffer, a 24 character window into the buffer, and keys for activating editing commands and for entering text; as illustrated in Figure B.1. The algebraic specification of the editor is given in Figure B.2.

One of the approaches compared in [121] was proposed by Guttag and Horning [122], which extend the algebraic axioms with routines which are specified using predicate transformers proposed by Dijkstra [123]. The idea here is that the axioms are more appropriate to specify abstract operations, while routines are better suited to specify concrete operations which are capable of handling system constraints (e.g. the limit of the buffer size). Figure B.2 provides the Guttag and Horning specification of the line editor.

Probably the most impressive work on data abstraction is the work carried out by Mallgren [37, 124]. This work was aimed at describing graphic data types to precisely describe graphical programming languages. One of the aspects of this research was describing interaction primitives whose semantics are precisely specified, making it possible to write and prove assertions about user interfaces and interactive programs. User interfaces necessitate the specification of concurrent operations, as it involves two parties - the human and the computer - which can attempt concurrent operations. Traditional algebraic specifications provide no facilities for the specification of concurrent operations. Mallgren developed an event algebra approach for the specification of concurrency. As described in [121],

> **an event algebra augments an existing algebra by defining a second algebra whose objects are states of the shared objects in the first algebra and whose operations are the state initialization, event, and characteristic functions associated with the shared objects and its operations in the first algebra.**

Mallgren views the user interface as containing a single shared object with associated operations for handling user and program interaction. As Mallgren [37] indicates, the resulting interface contains

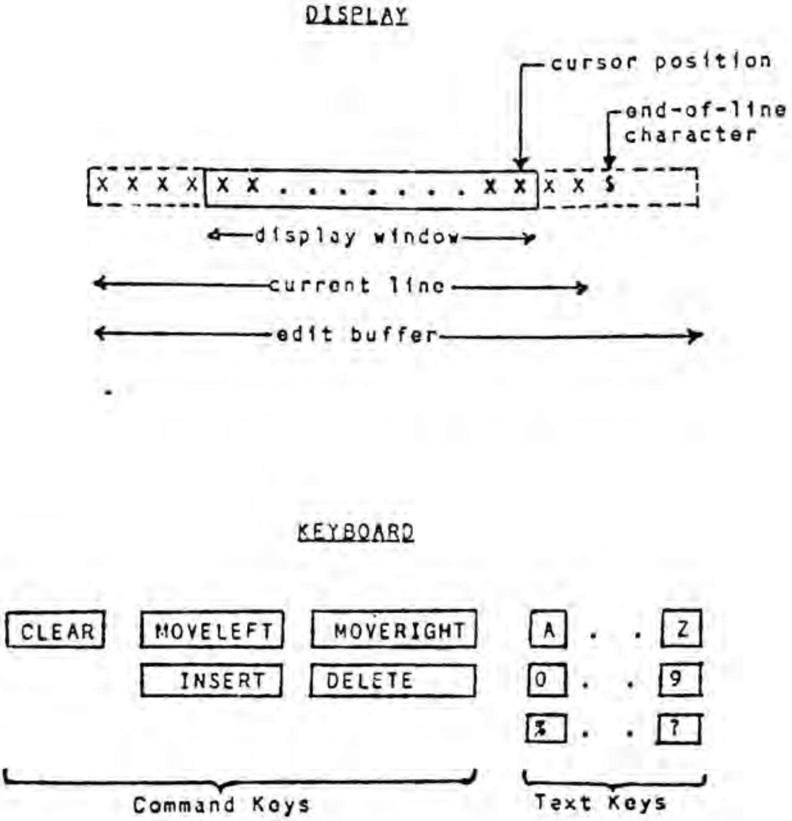


Figure B.1: The User Interface of a Line Editor

Generator Functions

```
1. clear: --> line
2. moveleft: line --> line
3. moveright: line --> line
4. insert: line x character x positive_integer --> line
5. delete: line x positive_integer --> line
6. record: line x {clr, move, insdel, type} --> line
```

Inquiry Functions

```
 7. curpos: line --> positive_integer
 8. size: line --> nonnegative_integer
 9. colchr: line --> character
10. content: line x positive_integer --> character
11. winstart: line --> positive_integer
```

Auxiliary Functions

```
12. maxsiz: --> positive_integer
13. winsiz: --> positive_integer
```

Axioms

```
1. curpos(emptyline) = 1
   size(emptyline) = 0
   colchr(emptyline) = ">"

 content(emptyline, i) = " "
 winstart(emptyline) = 1

 curpos(moveleft(L)) = curpos(L) - 1
 winstart(moveleft(L)) =
    IF winstart(L) > 1
    THEN winstart(L) - 1 ELSE winstart(L)

 curpos(moveright(L)) = curpos(L) + 1
 winstart(moveright(L)) =
    IF curpos(L) ≥ winsiz
    THEN winstart(L) + 1 ELSE winstart(L)

 size(insert(L, c, p)) = size(L) + 1
 content(insert(L, c, p)) = CASE
                              i < p: content(L, i)
                              i = p: c
                              i > p: content(L, i - 1)

 size(delete(L, p)) = size(L) - 1
 content(delete(L, p)) = CASE
                              i < p: content(L, i)
                              i ≥ p: content(L, i + 1)

 colchr(record(L, op)) = CASE op OF
                              clr: ">"
                              move: colchr(L)
                              insdel: IF colchr(L) = ">"
                                      THEN " " ELSE colchr(L)
                              type: "_"
```

Routines

```
 . PROCEDURE Clear(VAR line)
   SUCH THAT WP(Clear(L), Q) = Q[record(clear, clr)/L]

2. PROCEDURE Moveleft(VAR line)
   SUCH THAT WP(Moveleft(L), Q) =
     [     curpos(L) > 1 ==> Q[record(moveleft(L), move)/L]
     AND curpos(L) = 1 ==> Q[record(L, move)/L]              ]

3. PROCEDURE Moveright(VAR line)
   SUCH THAT WP(Moveright(L), Q) =
     [     (curpos(L) ≤ size(L) AND curpos(L) < maxsiz) ==>
                       .Q[record(moveright(L), move)/L]
     AND (curpos(L) > size(L) OR curpos(L) ≥ maxsiz) ==>
                        Q[record(L, move)/L]                 ]

4. PROCEDURE Insert(VAR line)
   SUCH THAT WP(Insert(L), Q) =
     [     (size(L) < maxsiz AND curpos(L) ≤ size(L)) ==>
           Q[record(insert(L, "[ ]", curpos(L)), insdel)/L]
     AND (size(L) = maxsiz AND curpos(L) ≤ size(L)) ==>
        Q[record(
             insert(delete(L, maxsiz), "[ ]", curpos(L)), insdel)/L]
     AND curpos(L) > size(L) ==>
        Q[record(L, insdel)/L]                               ]

5. PROCEDURE Delete(VAR line)
   SUCH THAT WP(Delete(L), Q) =
     [     (size(L) >  AND curpos(L) ≤ size(L)) ==>
        Q[record(delete(L, curpos(L)), insdel)/L]

     AND (size(L) = 0 OR curpos(L) > size(L)) ==>
                      Q[record(L, insdel)/L]                 ]

6. PROCEDURE Type(VAR line, char)
   SUCH THAT WP(Type(L, c), Q) =
     [     (curpos(L) < maxsiz AND curpos(L) ≤ size(L)) ==>
           Q[record(
              moveright(
                 insert(
                    delete(L, curpos(L)), c, curpos(L))), type)/L]
     AND (curpos(L) = maxsiz AND curpos(L) ≤ size(L)) ==>
        Q[record(
           insert(
              delete(L, curpos(L)), c, curpos(L)), type)/L]
     AND (curpos(L) < maxsiz AND curpos(L) > size(L)) ==>
        Q[record(moveright(insert(L, c, curpos(L))), type)/L]
     AND (curpos(L) = maxsiz AND curpos(L) > size(L)) ==>
        Q[record(insert(L,c,curpos(L)), type)/L]             ]

7. FUNCTION Display(VAR line, position)
   SUCH THAT PRE(Display(L, p)) = 1 ≤ p ≤ winsiz
   VALUE Display(L, p) =
     [     (winstart(L) + p - 1 < size(L) + 1)
                  ==> content(L, winstart(L) + p - 1)
     AND (winstart(L) + p - 1 = size(L) + 1 AND curpos(L) = size(L) + 1)
                  ==> colchr(L)
     AND (   (winstart(L) + p - 1 = size(L) + 1 AND curpos(L) < size(L) + 1)
           OR (winstart(L) + p - 1 > size(L) + 1))
                  ==> " "                                    ]

8. FUNCTION Blink(VAR line)
   VALUE Blink(L) =
     [     curpos(L) ≤ size(L) ==> curpos(L) - winstart(L) + 1
     AND curpos(L) > size(L) ==> NONE                        ]
```

Notes on Notation: $Q[x/y]$ stands for the predicate Q with x substituted for all free occurrences of y [10].

Figure B.2: Algebraic Specification of the Line Editor

> **the complete history of operation calls (and their parameters) since initialization. Thus, information is added to the interface whenever any of its operations is invoked. The interface provides information in two ways: by returning a value to an operation call and by controlling the time at which the operation returns.**

The event algebra specification of the line editor is presented in Figure B.3.

Program Actions

```
1. getkey: -> character
2. putline: line -->
```

User Actions

```
3. keystroke: character -->
4. readline: --> array_of_characters[1..winsiz]
```

Generator Functions (Event Functions of the Above Four Actions)

```
 5. $init: --> UIstate
 6. getkey$call: UIstate --> UIstate
 7. getkey$return: UIstate --> UIstate
 8. putline$call: UIstate x line --> UIstate
 9. putline$return: UIstate x line --> UIstate
10. keystroke$call: UIstate x character --> UIstate
11. keystroke$return: UIstate x character --> UIstate
12. readline$call: UIstate --> UIstate
13. readline$return: UIstate --> UIstate
```

Inquiry Operations (Characteristic Functions of the Four Actions)

```
14. getkey:wait: UIstate --> boolean
15. getkey:value: UIstate --> character
16. putline:wait: UIstate x line --> boolean
17. putline:value: UIstate x line -->
18. keystroke:wait: UIstate x character --> boolean
19. keystroke:value: UIstate x character -->
20. readline:wait: UIstate --> boolean
21. readline:value:UIstate -->
        array_of_characters[1..winsiz]
```

Axioms

```
1. getkey
  a. :wait($init) = true
  b. :wait(getkey$return(S)) = true
  c. :wait(keystroke$return(S, c)) = false
  d. :value($init) = undefined
  e. :value(keystroke$return(S, c)) =
        IF getkey:wait(S)
        THEN c
        ELSE :value(S)
2. readline
  a. :wait($init) = true
  b. :wait(putline$return(S, L)) = false
  c. :wait(readline$return(S)) = true
  d. :value($init) = undefined
  e. :value(putline$return(S, L)) =
        IF readline:wait(S)
        THEN display(L, 1)..display(L, winsiz)
        ELSE :value(S)
```

Sample User Interface Interaction

```
PROGRAM UI
VAR c: character; L: line;
BEGIN
L ← clear; putline(L); c <- getkey;
WHILE ¬(c = offkey) DO
  BEGIN
  IF c = enterkey
  THEN L <- interpret(L)
  ELSE CASE c OF

        clearkey: L <- clear
        leftkey: L <- moveleft(L)
        rightkey: L <- moveright(L)
        insertkey: L <- insert(L)
        deletekey: L <- delete(L)
        OTHERWISE: L <- type(L, c);
    putline(L);
    c <- getkey
    END
  END

  USER UI
  CONSTANT winsiz = 24;
  VAR c: character; r: array of characters[1..winsiz];
  BEGIN
  r <- readline;
  WHILE true DO
    BEGIN
    keystroke(c);
    r <- readline
    END
  END
```

Figure B.3: Event Algebra Specification of the Line Editor

APPENDIX C

THE MVC TRIAD:

LISTHOLDER, LISTHOLDERVIEW AND LISTHOLDERCONTROLLER

```
Model subclass: #ListHolder
        instanceVariableNames: 'list selection clipboard undoAction
                                             undoClipboard undoSelection oldentry '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Interface-Lists'
```

I am a model for manipulating lists represented as ordered collections.

```
ListHolder methodsFor: 'initialize-release'

initialize
        "initialize the ListHolder"

        selection ←0.
        clipboard ←'---'.
        undoClipboard ←nil.
        undoSelection ←nil.
        undoAction ←nil

ListHolder methodsFor: 'accessing'

list
        "Answer the list currently held by the receiver.
        (Message is sent by ListView«update:.)"

        ↑list

list: anOrderedCollection
        "Set anOrderedCollection to be what the receiver holds."

        list ←anOrderedCollection.
        selection ←0.
        self changed: #list

listIndex
        "Answer the index into the receiver's list of the current selection.
        (Message is sent by ListView«update:.)"

        ↑selection
```

```
toggleListIndex: anInteger
        "User selected or deselected anIntegerth element of the list --
        record it for use by ListHolder operations.
        (Message is sent by ListController»changeModelSelection: --
                therefore, its name is forced even though inaccurate.)"

        selection = anInteger
                ifTrue:  [selection ←0]
                ifFalse:[selection ←anInteger].
        self changed: #listIndex
```

ListHolder methodsFor: 'actions'

```
copySelection
        clipboard ←list at: selection

editSelection
        self undoAction: #restoreSelection.
        oldentry ←list at: selection.
        list at: selection
                put: (FillInTheBlank request: 'edit entry:'
                                        initialAnswer: (list at: selection)).
        self changed: #list

newAfter
        selection ←selection + 1.
                "turns out add:beforeIndex: works with index one past end"
        self newBefore

newBefore
        self undoAction: #removeSelection.
        list add: (FillInTheBlank request: 'new entry:') beforeIndex: selection.
        self changed: #list

pasteAfter
        selection ←selection + 1.
                "turns out add:beforeIndex: works with index one past end"
        self pasteBefore
```

```
pasteBefore
        self undoAction: #removeSelection.
        list add: clipboard beforeIndex: selection.
        self changed: #list

removeSelection
        self undoAction: #pasteBefore.
        self copySelection.              "put selection on clipboard"
        list removeAtIndex: selection.
        selection > list size ifTrue:[selection←list size].
        self changed: #list



restoreSelection
        "undo last entry edit or entry edit undo
         by swapping current entry and oldentry"

        |cur|
        cur←list at: selection.
        self undoAction: #restoreSelection.
        list at: selection put: oldentry.
        oldentry←cur.
        self changed: #list

undo
        "Undo the last action by executing the inverseAction block."

        | clip |
        undoAction = nil
                ifTrue:  [↑nil]
                ifFalse:[clip ←undoClipboard.
                        selection←undoSelection.
                        self perform: undoAction.
                        clipboard←clip]

undoAction: aSymbol
        "record undo selector plus current selection and clipboard"

        undoClipboard←clipboard.
        undoSelection←selection.
        undoAction←aSymbol
```

```
ListHolder class
        instanceVariableNames: ''
```

ListHolder class methodsFor: 'instance creation'

```
new
        ↑super new initialize

onList: aList
        "create a new ListHolder for aList"

        | aListHolder |
        aListHolder ← self new.
        aListHolder list: aList.
        ↑aListHolder
```

```
ListView subclass: #ListHolderView
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Interface-Lists'
```

I view a list stored in a ListHolder.

ListHolderView methodsFor: 'controller access'

```
defaultControllerClass
        ↑ListHolderController
```

ListHolderView methodsFor: 'updating'

```
update: aSymbol
        super update: aSymbol.
        aSymbol == #list
                ifTrue: [super update: #listIndex]
        "If list has been redisplayed, restore list index.
         Warning: redisplay moves list back to top;
         selection may be out of view."
```

```
ListHolderView class
        instanceVariableNames: ''

ListHolderView class methodsFor: 'instance creation'

openOn: aListHolder

        self openOn: aListHolder named: 'List'

openOn: aListHolder named: aString
        "schedule a List editor"

        | topView aListHolderView |


        "Create the list view."
        topView←StandardSystemView
                        model: aListHolder
                        label: aString
                        minimumSize: 100 @ 150.

        "Add the list view to the top view."
        aListHolderView←self new.
        aListHolderView model: aListHolder.
        aListHolderView insideColor: Form white.
        aListHolderView borderWidthLeft: 1 right: 1 top: 1 bottom: 1.

        "Add the list view to the top view."
        topView addSubView: aListHolderView.

        "Initialize the list view with the list."
        aListHolderView list: aListHolder list.

        "Schedule the top view's controller."
        topView controller open
```

```
ListController subclass: #ListHolderController
        instanceVariableNames: ''
        classVariableNames: 'ListHolderYellowButtonMenu
                                        ListHolderYellowButtonMessages '
        poolDictionaries: ''
        category: 'Interface-Lists'
```

I am used to manipulate ListHolders and their views.

ListHolderController methodsFor: 'initialize-release'

```
initialize
        super initialize.
        self initializeYellowButtonMenu.
        ↑self
```

ListHolderController methodsFor: 'private'

```
cantDo
        view flash flash

initializeYellowButtonMenu
        self yellowButtonMenu: ListHolderYellowButtonMenu
                yellowButtonMessages: ListHolderYellowButtonMessages

modelDo: aSymbol
        "send model the message aSymbol bracketed
         by controlTerminate/controlInitialize to avoid
         various scroll bar confusions"

        self controlTerminate.
        model perform: aSymbol.
        self controlInitialize
```

ListHolderController methodsFor: 'menu messages'

copySelection

```
        view selection = 0
                ifTrue: [self cantDo]
                ifFalse: [model copySelection]
```

cutSelection

```
        view selection = 0
                ifTrue:  [self cantDo]
                ifFalse: [self modelDo: #removeSelection]
```

editSelection

```
        view selection = 0
                ifTrue: [self cantDo]
                ifFalse: [self modelDo: #editSelection]
```

insertAfter

```
        view selection = 0
                ifTrue: [self cantDo]
                ifFalse: [self modelDo: #pasteAfter]
```

insertBefore

```
        view selection = 0
                ifTrue: [self cantDo]
                ifFalse: [self modelDo: #pasteBefore]
```

newAfter

```
        view selection = 0
                ifTrue: [self cantDo]
                ifFalse: [self modelDo: #newAfter]
```

newBefore

```
        view selection = 0
                ifTrue: [self cantDo]
                ifFalse: [self modelDo: #newBefore]
```

undo

```
        self controlTerminate
        "otherwise scroll bar gets confused --
         handled here instead of using modelDo:
         because need value of model message"

        model undo = nil
                ifTrue: [self cantDo].
        self controlInitialize
```

**ListHolderController class
        instanceVariableNames: ''**

<u>ListHolderController class methodsFor: 'class initialization'</u>

initialize

```
        "Initialize the menu for the yellow mouse button."
        "ListHolderController initialize"

        ListHolderYellowButtonMenu ←
                PopUpMenu labels:
'undo\edit\copy\cut\paste before\paste after\new before\new after' withCRs
                                        lines: #(1 4).
        ListHolderYellowButtonMessages ←
                #(undo editSelection copySelection cutSelection
                        insertBefore insertAfter newBefore newAfter)
```

<u>ListHolderController class methodsFor: 'instance creation'</u>

new

```
        ↑super new initialize
```

APPENDIX D

CLASS NOTEBOOK: USING SELECTIONINLIST

```
Model subclass: #Notebook
        instanceVariableNames: 'changed filename sections section
                                        entries entry '
        classVariableNames: 'Delimiter EntryDelimiter EntryMenu
                                NoEntryMenu NoSectionMenu
                                SectionDelimiter SectionMenu TextMenu '
        poolDictionaries: ''
        category: 'Interface-Notebook'
```

A generic two-level mechanism which organizes chunks of arbitrary text into named entries which are themselves organized into named sections. Uses would include phonebooks (category/person), notebooks (topic/title), and course catalogues (department/name).

Instance Variables:

sections	*dictionary of dictionaries*
section	*key of current section*
entries	*dictionary of current section* *(for convenience of entry methods)*
entry	*key of current entry in current section''s dictionary*

Notebook methodsFor: 'initialize-release'

```
initialize
        sections ←Dictionary new.
        section ←0.
        entry ←0.
        filename ←''.
        changed ←false
```

Notebook methodsFor: 'changing'

```
changeRequest
        "Is change, in particular closing the interface, OK?"

        changed ifTrue: [(self confirm: 'Changes have been made since last put;
do you really want to close?') ifFalse: [↑false]].
        ↑super changeRequest
```

Notebook methodsFor: 'sections'

```
addSection
        "Add a new section."

        |name|
        name ←FillInTheBlank request: 'Name of new section?' initialAnswer: ''.
        [sections includesKey: name] whileTrue:
                [name ←FillInTheBlank
                        request: 'Name of new section?' initialAnswer: name.
                (name = '') ifTrue: [↑self]                        "abort"
                ].
        [sections includesKey: name] whileTrue:
                [name ←FillInTheBlank
                        request: 'A section with that name already exists;
name of new section?'
                                        initialAnswer: name].
        name = '' ifTrue: [↑self].                 "abort"
        self addSection: name.
        self setSection: name.
        changed ←true.
        self changed: #section

currentSection
        ↑section

currentSection: aString
        self setSection: aString

removeSection
        "Remove current section, prompting for confirmation."

        (BinaryChoice message:
                'Do you really want to remove section
"', section, '"?') ifTrue:
                        [sections removeKey: section.
                        self setSection: nil.
                        changed ←true.
                        self changed: #section]
```

```
renameSection
        "Rename current section."

        |newname|
        newname←FillInTheBlank
                        request: 'New name for section
"',section,'"?'
                        initialAnswer: section.
        newname = section ifTrue: [↑self].                  "abort if unchanged"
        [sections includesKey: newname] whileTrue:
                [newname←FillInTheBlank
                                request: 'A section with that name already exists;
new name for "',section,'"?'
                                        initialAnswer: newname].
        newname = '' | newname = section ifTrue: [↑self].               "abort"
        sections at: newname put: (sections at: section).
        sections removeKey: section.
        self setSection: newname.
        changed←true.
        self changed: #section

sectionMenu
        "the menu of section commands"

        section == nil
                ifTrue: [↑NoSectionMenu]
                ifFalse: [↑SectionMenu]

sectionNames
        ↑sections keys asSortedCollection
```

Notebook methodsFor: 'entries'

```
addEntry
        "Add a new entry."

        |name|
        name ←FillInTheBlank request: 'Name of new entry?' initialAnswer: ''.
        [entries includesKey: name] whileTrue:
                [name ←FillInTheBlank
                                request: 'Name of new entry?' initialAnswer: name.
                (name = '') ifTrue: [↑self]                       "abort"
                ].
        [entries includesKey: name] whileTrue:
                [name ←FillInTheBlank
                        request: 'A entry with that name already exists;
name of new entry?'
                                        initialAnswer: name].
        name = '' ifTrue: [↑self].                                "abort"
        self addEntry: name to: entries.
        self setEntry: name.
        changed ←true.
        self changed: #entry

currentEntry
        ↑entry

currentEntry: aString
        self setEntry: aString

entryMenu
        "the menu of entry commands"

        entry == nil
                ifTrue: [↑NoEntryMenu]
                ifFalse: [↑EntryMenu]

entryNames
        entries == nil ifTrue: [↑nil].
        ↑entries keys asSortedCollection
```

```
removeEntry
        "Remove current entry, prompting for confirmation."

        (BinaryChoice message:
                'Do you really want to remove entry "', entry, '"?') ifTrue:
                        [entries removeKey: entry.
                        self setEntry: nil.
                        changed←true.
                        self changed: #entry]
```

```
renameEntry
        "Rename current entry."

        |newname|
        newname←FillInTheBlank request: 'New name for entry
"',entry,'"?'
                                                  initialAnswer: entry.
        newname = entry ifTrue: [↑self].                        "abort if unchanged"
        [entries includesKey: newname] whileTrue:
                [newname ←FillInTheBlank
                                request: 'A entry with that name already exists;
new name for "',entry,'"?'
                                initialAnswer: newname].
        newname = '' | newname = entry ifTrue: [↑self].                    "abort"
        entries at: newname put: (entries at: entry).
        entries removeKey: entry.
        self setEntry: newname.
        changed←true.
        self changed: #entry
```

<u>Notebook methodsFor: 'text'</u>

```
acceptText: aText
        "Enter aText at current entry."

        entry == nil ifTrue: [↑false].
        self put: aText copy at: entry in: entries.
        changed←true.
        ↑true
```

```
text
        "the text of the current entry"

        entry == nil ifTrue: [↑nil].
        ↑entries at: entry

textMenu
        ↑TextMenu
```

Notebook methodsFor: 'private'

```
addEntry: aString to: aDictionary
        "Add a new section named aString"

        aDictionary at: aString put: Text new

addSection: aString
        "Add a new section named aString"

        sections at: aString put: Dictionary new

put: aString at: anEntryName in: aDictionary
        "Make aString be the current text for anEntryName in aDictionary."

        aDictionary at: anEntryName put: aString

setEntry: aString
        "Make aString be the current entry."

        entry ←aString.
        self changed: #text

setSection: aString
        "Make aString be the current section."

        section ←aString.
        section == nil
                ifTrue: [entries ←nil]
                ifFalse: [entries ←sections at: aString].
        self setEntry: nil.
        self changed: #entry
```

Notebook methodsFor: 'fileIn/Out'

```
get
        "Add entries from notebook stored on a file."

        | filnam |

        filnam←(FillInTheBlank request: 'Name of file to get notebook from?'
                                        initialAnswer: filename).
        self getFrom: filnam.           "Should check to see if it exists"
        self changed #section

getEntryIn: aDictionary From: aStream
        "Read entry from stream.
        The stream starts positioned at first character of entry name"

        |ent text|

        ent←aStream upTo: Character cr.
        Transcript show: '   ',ent; cr.
        text←(aStream upTo: Delimiter) asText.      "upTo: stops normally at EOF"
        self addEntry: ent to: aDictionary.
        self put: text at: ent in: aDictionary

getFrom: aFileName
        "add entries from notebook file to this notebook, overriding any
        text already stored for duplicated section/entry pairs."

        |strm|

        filename = '' ifTrue: [filename←aFileName].
                                        "If getting several, keep first name"
        strm←(FileStream oldFileNamed: aFileName).
        strm skipTo: Delimiter; next   "skip any header info in file"

        [strm atEnd] whileFalse:
                [self getSectionFrom: strm].

        strm close
```

```
getSectionFrom: aStream
        "Read section and its entries from stream.
        The stream starts positioned at first character of section name"

        |sec secdict|

        sec←aStream upTo: Character cr.
        aStream next.                           "gobble next delimiter"
        Transcript show: sec; cr.
        (sections includesKey: sec) ifFalse: [self addSection: sec].
        secdict←sections at: sec.
        [aStream atEnd | (aStream peekFor: Delimiter)] whileFalse:
                [self getEntryIn: secdict From: aStream]

put
        "File notebook out putting two delimiter characters in front of section
        heads and one in front of entry heads.  Delimiter character cannot
        be used in headings or entry text."

        | strm |

        strm←FileStream fileNamed:          "Should check to see if it exists"
                (filename←(FillInTheBlank
                                request: 'Name of file to put notebook to?'
                                initialAnswer: filename)).
        self putTo: strm.
        changed←false

putEntry: anAssoc on: aStream
        "file out the entry named aName on aStream"

        |text|

        text←anAssoc value.

        aStream nextPut: Delimiter;
                nextPutAll: anAssoc key; cr;
                nextPutAll: text.
        text isEmpty ifFalse:
                [(text last = Character cr) ifFalse:
                        [aStream cr]]
```

```
putSection: anAssoc on: aStream
        "file out the section named aName on aStream"

        aStream         nextPut: Delimiter; nextPut: Delimiter;
                        nextPutAll: anAssoc key; cr.
        anAssoc value associationsDo: [:assoc | self putEntry: assoc on: aStream]
                        "Dictionary enumeration is arbitrary order;
                         if want alphabetical, must sort keys and access each."


putTo: aStream
        "File notebook out using markers for section heads and entry heads"

        aStream nextPut: Character cr.          "could put a header here"
        sections associationsDo: [:sec | self putSection: sec on: aStream].
                        "Dictionary enumeration is arbitrary order;
                         if want alphabetical, must sort keys and access each."
        aStream shorten; close
```

Notebook class methodsFor: 'class initialization'

```
initialize
        "Initialize Notebook with appropriate menus and fileIn/Out delimiters."

        "Notebook initialize"

        Delimiter←$\.

        SectionMenu←ActionMenu
                labels: 'add section\put to file\load from file\rename\remove'
                        withCRs
                lines: #(3)
                selectors: #(addSection put get renameSection removeSection).
        NoSectionMenu←ActionMenu
                labels: 'add section\put to file\load from file'
                selectors: #(addSection put get).
        EntryMenu←ActionMenu
                labels: 'add entry\rename\remove' withCRs
                lines: #(1)
                selectors: #(addEntry renameEntry removeEntry).
        NoEntryMenu←ActionMenu
                labels: 'add entry'
                selectors: #(addEntry).
        TextMenu←ActionMenu
                labels: 'again\undo\copy\cut\paste\accept\cancel' withCRs
                lines: #(2 5)
                selectors: #(again undo copySelection cut paste accept cancel)
```

Notebook class methodsFor: 'instance creation'

```
from: aFileName
        "Create a new notebook and get its contents from aFileName."

        | notebook |
        notebook ←self new.
        notebook getFrom: aFileName.
        ↑notebook

new
        ↑super new initialize
```

Notebook class methodsFor: 'user interface'

```
openFrom: aFileName
        "Open an MVC interface on a new Notebook with contents from
        aFileName."

        self openOn: (Notebook from: aFileName) named: aFileName

openNamed: aString
        "Open an MVC interface on a new Notebook."

        "Notebook openNamed: 'Notebook'"

        self openOn: Notebook new named: aString


openOn: aNotebook named: aString
      "Open an MVC interface on aNotebook.
      This could really be an instance method, but it is conventional
      to make MVC creators class methods."


      "Style note: in general methods should be smaller than a page and do
       just one thing, so I would normally break this up into subfunctions.
      However, in this case I felt it would confuse the presentation.
      Also, I wanted this method to look like typical system MVC interface
      creation methods, which typically are in one piece."
```

```
| topView sectionView entryView textView |

topView←StandardSystemView
                model: aNotebook
                label: aString
                minimumSize: 275@350.

sectionView←SelectionInListView
        on: aNotebook                       "model"
        aspect: #section                    "change symbol for section"
        change: #currentSection:            "msg to select new section"
        list: #sectionNames                 "msg to get list of sections"
        menu: #sectionMenu                  "msg to get section menu"
        initialSelection: #currentSection.  "msg to get initial section"

entryView←SelectionInListView
        on: aNotebook                       "model"
        aspect: #entry                      "change symbol for entry"
        change: #currentEntry:              "msg to select new entry"
        list: #entryNames                   "msg to get list of entries"
        menu: #entryMenu                    "msg to get entry menu"
        initialSelection: #currentEntry.    "msg to get initial entry"

textView←TextView
        on: aNotebook                       "model"
        aspect: #text                       "change symbol for new text"
        change: #acceptText:                "msg to store new entry text"
        menu: #textMenu.                    "msg for text menu"

topView addSubView: sectionView
                in: (0@0 extent: 1@0.2)"top 20%"
                borderWidth: 1.

topView addSubView: entryView
                in: (0@0.2 extent: 1@0.25)     "next 25%"
                borderWidth: 1.

topView addSubView: textView
                in: (0@0.45 extent: 1@0.55)    "bottom 55%"
                borderWidth: 1.

"There are many other messages for adding subviews;
addSubView:in:borderWidth: happens to be very convenient
for arranging multiple subviews."

topView controller open                     "activate the whole interface"
```

BIBLIOGRAPHY

[1] Teitelbaum, Tim and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," **Communications of the ACM,** vol. 29, no. 9, pp. 563-573, Sept. 1981.

[2] Moran, Thomas P., "Guest Editor's Introduction: An Applied Psychology of the User," **Computing Surveys,** vol. 13, no. 1, pp. 1-11, ACM, New York, March 1981.

[3] Newell, Martin E., "Towards a Design Methodology for Interactive Systems," in **Methodology of Interaction,** ed. R. A. Guedj et al., pp. 317-324, North-Holland Publishing Company, Amsterdam, Netherlands, 1980.

[4] Nievergelt, Jurg, "Geometric Designs," Colloquium talk at the University of Central Florida, November 1985.

[5] Kay, Alan and Adele Goldberg, "Personal Dynamic Media," **Computer,** pp. 31-41, IEEE, March 1977.

[6] Nickerson, R. S., "On Conversational Interaction with Computers," in **User-Oriented Design of Interactive Graphics Systems,** ed. Sigfried Treu, Proceedings of the ACM/SIGGRAPH Workshop, pp. 101-113, ACM, Pittsburgh, PA., October 1976.

[7] Baecker, Ronald, "Towards an Effective Characterization of Graphical Interaction," in **Methodology of Interaction,** ed. R. A. Guedj et al., pp. 127-147, North-Holland Publishing Company, Amsterdam, Netherlands, 1980.

[8] Guedj, R. A., **Methodology of Interaction,** North-Holland Publishing Company, Amsterdam, Netherlands, 1980.

[9] Tozzi, Clesio, "Man-Machine Communication in Process Control," in **Methodology of Interaction,** ed. R. A. Guedj et al., pp. 393-398, North-Holland Publishing Company, Amsterdam, Netherlands, 1980.

[10] Ohsuga, Setsuo, "Towards Intelligent Interactive Systems," in **Methodology of Interaction,** ed. R. A. Guedj et al., pp. 339-360, North-Holland Publishing Company, Amsterdam, Netherlands, 1980.

[11] Dzida, W., S. Herda, and W. D. Itzfeldt, "User-Perceived Quality of Interactive Systems," in **Methodology of Interaction,** ed. R. A. Guedj et al., pp. 189-193, North-Holland Publishing Company, Amsterdam, Netherlands, 1980.

[12] Foley, J. D. and V. L. Wallace, "The Art of Natural Graphic Man-Machine Communication," **Proceedings of the IEEE,** vol. 62, pp. 462-471, IEEE, April 1974.

[13] Foley, J. D., "The Structure of Command Languages," in **Methodology of Interaction,** ed. R. A. Guedj et al., pp. 227-234, North-Holland Publishing Company, Amsterdam, Netherlands, 1980.

[14] Foley, J. D., V. L. Wallace, and P. Chan, "The Human Factors of Computer Graphics Interaction Techniques," **Computer Graphics and Applications,** vol. 4, no. 11, pp. 13-48, IEEE, Nov. 1984.

[15] Britton, E. G., "A Methodology for the Ergonomic Design of Interactive Computer Graphics Systems," Ph.D. Dissertation, Dept. of Computer Science, Univ. of North Carolina, Chapel Hill, NC., 1977.

[16] Wallace, V. L., "Conversational Ergonomics," in **Workshop on User Oriented Design of Interactive Graphics Systems,** ACM, Pittsburgh, PA., Oct. 1976.

[17] Newman, W. M, and R. F. Sproull, **Principles of Interactive Computer Graphics,** McGraw-Hill, New York, NY., 1979.

[18] Moran, Thomas P., "Introduction to the Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems," Report SSL-78-3, Xerox Palo Alto Research Center, Palo Alto, CA., 1978.

[19] Dunn, R. M., "A Philosophical Prelude to Methodology of Interaction," in **Methodology of Interaction,** ed. R. A. Guedj et al., pp. 183-188, North-Holland Publishing Company, Amsterdam, Netherlands, 1980.

[20] Hornbuckle, G. D., "The Computer Graphics/User Interface," **IEEE Trans. Human Factors In Electronics,** vol. HFE-8, no. 1, pp. 17-22, IEEE, March 1967.

[21] Hansen, W. J., "User Engineering Principles for Interactive Systems," in **Proceedings Fall Joint Computer Conference,** vol. 39, pp. 523-532, AFIPS Press, Arlington, VA., 1971.

[22] Green, M., "A Methodology for the Specification of Graphical User Interfaces," **Computer Graphics,** vol. 15, no. 3, pp. 99-108, ACM, Aug. 1981.

[23] Robinson, L., "The HDM Handbook, Volume 1: The Foundations of HDM," Project Report 4828, Computer Science Group, SRI International, 1979.

[24] Moran, Thomas P., "The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems," **International Journal of Man-Machine Studies,** vol. 15, pp. 3-50, 1981.

[25] Minsky, M., "A Framework for Representing Knowledge," in **The Psychology of Computer Vision,** ed. P. H. Winston, pp. 211-277, McGraw-Hill, New York, NY., 1975.

[26] Bobrow, D. G. and T. Winograd, "An Overview of KRL, A Knowledge Representation Language," **Cognitive Science,** vol. 1, pp. 3-46, 1977.

[27] Norman, D. A., et al., **Explorations In Cognition,** W. H. Freeman, San Francisco, CA., 1975.

[28] Brachman, R. J., "What's in a Concept: Structural Foundations for Semantic Networks," **Internation Journal of Man-Machine Studies,** vol. 9, pp. 127-152, 1977.

[29] Newell, A. and H. A. Simon, **Human Problem Solving,** Prentice-Hall, Englewood Cliffs, NJ., 1972.

[30] Brooks, F. P., "No Silver Bullet: Essence and Accidents of Software Engineering," **Computer,** vol. 20, pp. 10-19, 1987.

[31] Cox, B. J., **Object Oriented Programming: An Evolutionary Approach,** Addison-Wesley, Reading, Mass., 1986.

[32] Flynn, M. J., "Very High Speed Computing Systems", **Proceedings of the IEEE,** vol. 54, pp. 1901-1909, December 1966.

[33] Pfaff, G. E., Ed., **User Interface Management Systems,** Springer-Verlag, Berlin, 1985.

[34] Liskov, B. and S. Zilles, "An Introduction to Formal Specification of Data Abstractions," in **Current Trends In Programming Methodology, Volume 1: Software Specification and Design,** ed. R. T. Yeh, pp. 1-32, Prentice-Hall, Englewood Cliffs, NJ., 1977.

[35] Liskov, B. and S. Zilles, "Specification Techniques for Data Abstractions," **IEEE Trans. Software Engineering,** vol. SE-1, no. 1, pp. 7-19, March 1975.

[36] Guttag, J. V. and J. J. Horning, "The Algebraic Specification of Abstract Data Types," **Acta Informatica,** vol. 100, no. 1, pp. 27-52, 1978.

[37] Mallgren, W. R., "Formal Specification of Graphic Data Types," **ACM Trans. Programming Languages and Systems,** vol. 4, no. 4, pp. 687-710, Oct. 1982.

[38] Jacob, R. J. K., "Using Formal Specifications in the Design of a Human-Computer Interface," **Communications of the ACM,** vol. 26, no. 4, pp. 259-264, April 1983.

[39] Shneiderman, B., "Multiparty Grammars and Related Features for Defining Interactive Systems," **IEEE Trans. Systems, Man, and Cybernetics,** vol. SMC-12, no. 2, pp. 148-154, March/April 1982.

[40] van den Bos, J., "Definition and Use of Higher-level Graphics Input Tools," in SIGGRAPH '78 Proceedings, **ACM Computer Graphics,** vol. 12, no. 3, pp. 38-42, Aug. 1978.

[41] van den Bos, J., M. J. Plasmeijer, and P. H. Hartel, "Input-output Tools: A Language Facility for Interactive and Real-time Systems," **IEEE Trans. Software Engineering,** vol. SE-9, no. 3, pp. 247-259, 1983.

[42] Borufka, H. G., H. W. Kuhlmann and P. J. W. ten Hagen, "Dialogue Cells: A Method for Defining Interactions," **IEEE Computer Graphics and Applications,** vol. 2, no. 5, pp. 25-33, 1982.

[43] ten Hagen, P. J. W. and J. Derksen, "Parallel Input and Feedback in Dialogue Cells," in **User Interface Management Systems,** ed. G. E. Pfaff, pp. 109-124, Springer-Verlag, Berlin, 1985.

[44] Parnas, D. L., "On the Use of Transition Diagrams in the Design of a User Interface for an Interactive Computer System," in **Proceedings 24th National ACM Conference,** pp. 379-385, ACM, New York, NY., 1969.

[45] Aho, A. V., R. Sethi and J. D. Ullman, **Compilers, Principles, Techniques, and Tools,** Addison-Wesley, Reading, Mass., 1986.

[46] Shaw, A. C., "On the Specification of Graphics Command Languages and their Processors," in **Methodology of Interaction,** ed. R. A. Guedj et al., pp. 377-392, North-Holland Publishing Company, Amsterdam, Netherlands, 1980.

[47] Shaw, A. C., "Systems Design and Documentation Using Path Descriptions," in **Proceedings 1975 Sagamore Computer Conference on Parallel Processing,** pp. 180-181, IEEE, Silver Spring, MD., 1975.

[48] Shaw, A. C., "Software Descriptions with Flow Expressions," **IEEE Trans. Software Engineering,** vol. SE-4, no. 3, pp. 242-254, May 1978.

[49] MUMPS Development Committee, **MUMPS Language Standard,** American National Standards Institute, New York, NY., 1977.

[50] Singer, A., "Formal Methods and Human Factors in the Design of Interactive Languages," Ph.D. Dissertation, Dept. of Computer and Information Science, Univ. of Massachusetts, Amherst, MA., Sept. 1979.

[51] Green, M., "A Graphical Input Programming System," M.Sc. thesis, Dept. of Computer Science, Univ. of Toronto, Toronto, Canada, 1979.

[52] Green, M., "A Survey of Three Dialogue Models," **ACM Trans. on Graphics,** vol. 5, no. 3, pp. 244-275, July 1986.

[53] Newman, W. M., "A System for Interactive Graphical Programming," in **Proceedings of the Spring Joint Computer Conference,** pp. 47-54, Washington, DC., 1968.

[54] Edmonds, E. A. and S. P. Guest, "An Interactive Tutorial System for Teaching Programming," in **Proceedings of the IERE Conference 36: Computer Systems and**

Technology, pp. 263-270, IERE, London, 1977.

[55] Hanau, P. R. and D. R. Lenorovitz, "Prototyping and Simulation Tools for User/Computer Dialogue Design," in SIGGRAPH '80 Proceedings, **ACM Computer Graphics,** vol. 14, no. 3, pp. 271-278, July 1980.

[56] Olsen, D. R., "Automatic Generation of Interactive Systems," **ACM Computer Graphics,** vol. 17, no. 1, pp. 53-57, ACM, Jan. 1983.

[57] Olsen, D. R. and E. P. Dempsey, "Syntax Directed Graphical Interaction," **SIGPLAN Notices,** vol. 18, no. 6, pp. 112-117, ACM, June 1983.

[58] Olsen, D. R. and E. P. Dempsey, "SYNGRAPH: A Graphical User Interface Generator," **ACM Computer Graphics,** vol. 17, no. 3, pp. 43-50, July 1983.

[59] Denert, E., "Specification and Design of Dialogue Systems with State Diagrams," in **Proceedings of the International Computing Symposium,** pp. 417-424, North-Holland Publishing Company, Amsterdam, Netherlands, 1977.

[60] Casey, B. E. and B. Dasarathy, "Modeling and Validating the Man-Machine Interface," **Software Practice and Experience,** vol. 12, pp. 557-569, 1982.

[61] Feldman, M. B. and G. T. Rogers, "Toward the Design and Development of Style-Independent Interactive Systems," in **Proceedings of the ACM SIGCHI Human Factors in Computer Systems Conference,** pp. 111-116, ACM, New York, NY., 1982.

[62] Guest, S. P., "The Use of Software Tools for Dialogue Design," **International Journal Man-Machine Studies,** vol. 16, pp. 263-285, 1982.

[63] Kamran, A. and M. B. Feldman, "Graphics Programming Independent of Interaction Techniques and Styles," **ACM Computer Graphics,** vol. 17, no. 1, pp. 58-66, 1983.

[64] Kieras, D. and P. G. Polson, "A Generalized Transition Network Representation for Interactive Systems," in **Proceedings of the CHI '83 Human Factors in Computing Systems,** pp. 103-106, ACM, New York, NY., 1983.

[65] Schulert, A. J., G. T. Rogers and J. A. Hamilton, "ADM - A Dialogue Manager," in **Proceedings of the CHI '83 Human Factors in Computing Systems,** pp. 177-183, ACM, New York, NY., 1985.

[66] Sibert, J., R. Belliardi and A. Kamran, "Some Thoughts on the Interface Between User Interface Management Systems and Application Software," in **Methodology of Interaction,** ed. R. A. Guedj et al., pp. 183-192, North-Holland Publishing Company, Amsterdam, Netherlands, 1980.

[67] Wasserman, A. I., "Extending State Transition Diagrams for the Specification of Human-Computer Interactions," **IEEE Trans. Software Engineering,** vol. SE-11, no. 8, pp. 699-713, Aug. 1985.

[68] Jacob, R. J. K., "A Specification Language for Direct-Manipulation User Interfaces," **ACM Trans. on Graphics,** vol. 5, no. 4, pp. 283-317, Oct. 1986.

[69] Elshoff, E. L., et al., "Handling Asynchronous Interrupts in a PL/1-like language," **Software Practice and Experience,** vol. 4, pp. 117-124, 1974.

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

[71] Green, M., "The University of Alberta User Interface Management System," in SIGGRAPH '85 Proceedings, **ACM Computer Graphics,** vol. 19, no. 3, pp. 205-213, July 1985.

[72] Hill, R. D., "Supporting Concurrency, Communications and Synchronization in Human-Computer Interaction - The Sassafras User Interface Management System," **ACM Trans.**

on Graphics, vol. 5, no. 3, pp. 179-210, July 1986.

[73] Flecchia, M. A. and R. D. Bergeron, "Specifying Complex Dialogues in ALGEA," in **Proceedings of CHI and Graphics Interface '87,** pp. 222-228, Toronto, Canada, April 1987.

[74] "Status Report of the Graphics Standards Committee," **ACM Computer Graphics,** vol. 13, no. 3, Aug. 1979.

[75] International Standards Organization, **Graphical Kernel System (GKS), Version 6.6,** May 1981.

[76] Scheifler, R. W. and J. Gettys, "The X Window System," **ACM Trans. on Graphics,** vol. 5, no. 2, pp. 132-165, April 1986.

[77] Schmucker, K. J., **Object-Oriented Programming for the Macintosh,** Hayden Book Company, Hasbrouck Heights, NJ., 1986.

[78] Krasner, G. E. and S. T. Pope, "A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-80," ParcPlace Systems, 2400 Geng Road, Palo Alto, CA 94303, January 1988.

[79] Goldberg, A. and D. Robson, **Smalltalk-80: The Language and its Implementation,** Addison-Wesley, Reading, Mass., 1983.

[80] Wong, P. C. S. and E. R. Reid, "FLAIR - User Interface Dialog Design Tool," **ACM Computer Graphics,** vol. 16, no. 3, pp. 87-98, July 1982.

[81] Olsen, D. R., "MIKE: The Menu Interaction Kontrol Environment," **ACM Trans. on Graphics,** vol. 5, no. 4, pp. 318-344, Oct. 1986.

[82] Bournique, R. F., "User-Oriented Features and Language-Based Agents: A Study of Graphical Interaction Language Specification," Ph.D. Dissertation, Dept. of Computer Science, Univ. of Pittsburgh, PA., 1981.

[83] Reiss, S. P., "Graphical Program Development with PECAN Program Development Systems," **ACM SIGPLAN Notices,** vol. 19, no. 5, pp. 30-41, May 1984.

[84] Foley, J. D. and A. van Dam, **Fundamentals of Interactive Computer Graphics,** Addison-Wesley, Reading, Mass., 1982.

[85] Workman, D. A., F. Arefi and M. H. Dodani, "GRIP: A Formal Framework for Developing a Support Environment for Graphical Interactive Programming," **Proceedings of the Conference on Software Tools,** pp. 138-153, IEEE Computer Society, New York, NY., 1985.

[86] Arefi, Farahangiz, "Automatically Generating Syntax-Directed Editors for Graphical Languages," Ph.D. Dissertation, Dept. of Computer Science, Univ. of Central Florida, 1988.

[87] Goldberg, A., **Smalltalk-80: The Interactive Programming Environment,** Addison-Wesley, Reading, Mass., 1984.

[88] "Special Issue on Object-Oriented Programming," **BYTE,** August 1986.

[89] OOPSLA '86 Conference Proceedings, **ACM SIGPLAN Notices,** vol. 21, no. 10, 1986.

[90] OOPSLA '87 Conference Proceedings, **ACM SIGPLAN Notices,** vol. 22, no. 12, 1987.

[91] OOPSLA '88 Conference Proceedings, **ACM SIGPLAN Notices,** vol. 23, no. 11, 1988.

[92] Kernighan, B. W. and D. M. Ritchie, **The C Programming Language,** Prentice-Hall, Englewood Cliffs, NJ., 1987.

[93] Eckart, D. J., "Iteration and Abstract Data Types," **ACM SIGPLAN Notices,** vol. 22, no. 4, pp. 103-110, April 1987.

[94] Stroustrup, B., **The C++ Programming Language,** Addison-Wesley, Reading, Mass., 1986.

[95] Johnson, R. E. and B. Foote, "Designing Reusable Classes," **Journal of Object-Oriented Programming,** vol. 1, no. 2, pp. 22-35, June/July 1988.

[96] Cardelli, L. and P. Wegner, "On Understanding Types, Data Abstraction and Polymorphism," **ACM Computing Surveys,** vol. 17, no. 4, pp. 471-523, Dec. 1985.

[97] Booch, G., **Software Components with Ada,** Addison-Wesley, Reading, Mass., 1987.

[98] Goldstein, I. P. and D. G. Bobrow, "PIE: An Experimental Personal Information Environment," Technical Report CSL-81-4, Xerox Palo Alto Research Center, Palo Alto, CA., 1981.

[99] Alexander, J. H., "Painless Panes for Smalltalk Windows," in OOPSLA '87 Conference Proceedings, **ACM SIGPLAN Notices,** vol. 22, no. 12, pp. 287-294, Dec. 1987.

[100] Smith, R. B., "Experiences with the Alternative Reality Kit: An Example of the Tension Between Literalism and Magic," **IEEE Computer Graphics and Applications,** pp. 42-50, Sept. 1987.

[101] Kernighan, B. W. and R. Pike **The UNIX Programming Environment,** Prentice-Hall, Englewood Cliffs, NJ., 1984.

[102] Starbase Graphics Reference Manual, Hewlett Packard Company, Fort Collins, CO., 1987.

[103] MetaWindow Reference Manual, MetaGraphics Software Corp., Scotts Valley, CA., 1986.

[104] Domain Graphics Reference Manual, Appolo Computer Corp., 1987.

[105] Birkhead, E., "X11 Toolkit Extensions," **DEC Professional,** pp. 92-95, July 1988.

[106] Aho, A. V., J. E. Hopcroft and J. D. Ullman, **Data Structures and Algorithms,** Addison-Wesley, Reading, Mass., 1983.

[107] Beck, K., "Smalltalk Programming," **HOOPLA,** vol. 1, no. 1, pp. 4-12, OOPSTAD, Everett, WA., Feb. 1988.

[108] Adams, S. S., "MetaMethods: The MVC Paradigm," **HOOPLA,** vol. 1, no. 4, pp. 3-21, OOPSTAD, Everett, WA., July 1988.

[109] Smalltalk/V Mac Tutorial and Programming Handbook, Digitalk Inc., Los Angeles, CA., Sept. 1988.

[110] Apple Computer, **Inside Macintosh,** Addison-Wesley, Reading, Mass., 1985.

[111] Workman, D. A., "GRASP: A Software Development System Using D-Charts," **Software Practice and Experience,** vol.. 13, no. 1, pp. 17-32, 1983.

[112] Reps, T., "Generating Language-Based Environments," Ph.D. Dissertation, Dept. of Computer Science, Cornell University, Ithaca, NY., Oct. 1983.

[113] Borning, A., "The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory," **ACM TOPLAS,** vol. 3, no. 4, pp. 353-387, Oct. 1981.

[114] Duisberg, R., "Constraint-Based Animation: The Implementation of Temporal Constraints in the Animus System," Ph.D. Dissertation, Computer Science Department, Univ. of Washington, Seattle, WA., 1986.

[115] Mittal, S., C. L. Dym and M. Morjaria, "PRIDE: An Expert System for the Design of Paper Handling Systems," **IEEE Computer,** pp. 102-114, July 1986.

[116] Henry, R. and P. Damron, "Code Generation Using Tree Pattern Matchers," Technical Report 87-02-04, Computer Science Department, Univ. of Washington, Seattle, WA., 1987.

[117] Levitt, D., "Machine Tongues X: Constraint Languages," **Computer Music Journal,** vol. 8, no. 1, pp. 9-21, Spring 1984.

[118] Ege, R. K., D. Maier and A. Borning, "The Filter Browser: Defining Interfaces Graphically," in **Proceedings of the European, Conference on Object Oriented Programming,** Springer-Verlag Lecture Notes in Computer Science No. 276, Paris, France, June 1987.

[119] Borning, A. and R. Duisberg, "Constraint-Based Tools for Building User Interfaces," **ACM Trans. on Graphics,** vol. 5, no. 4, pp. 345-374, Oct. 1986.

[120] Bolour, A., et al., "The Role of Time in Information Processing: A Survey," **ACM SIGMOD Record,** vol. 12, no. 3, pp. 27-50, 1982.

[121] Chi, U. H., "Formal Specification of User Interfaces: A Comparison and Evaluation of Four Axiomatic Approaches," **IEEE Trans. Software Engineering,** vol. SE-11, no. 8, pp. 671-685, Aug. 1985.

[122] Guttag, J. V. and J. J. Horning, "Formal Specification as a Design Tool," in **Proceedings 7th Symposium Principles of Programming Languages,** pp. 251-261, ACM, New York, NY., 1980.

[123] Dijkstra, E. W., **A Discipline of Programming,** Prentice-Hall, Englewood Cliffs, NJ., 1976.

[124] Mallgren, W. R., "Formal Specification of Interactive Graphics Programming Languages," Ph.D. Dissertation, Dept. of Computer Science, Univ. of Washington, Seattle, WA., Sept. 1981.