5-21-1985

Application Interface Development Environment

Robert F. Gordon Ph.D.
Molloy College, rfgordon@molloy.edu

Barry E. Willner

RC 11160 (#50246) 5/21/85
Computer Science 20 pages

Research Report

Application Interface Development Environment

Robert F. Gordon

Barry E. Willner

IBM Thomas J. Watson Research Center

P.O. Box 218

Yorktown Heights, N.Y. 10598

IBM Research Division
San Jose · Yorktown · Zurich

Copies may be requested from:

IBM Thomas J. Watson Research Center
Distribution Services F-11 Stormytown
Post Office Box 218
Yorktown Heights, New York 10598

RC 11160 (#50246) 5/21/85
Computer Science 20 pages

Application Interface Development Environment

Robert F. Gordon
Barry E. Willner

IBM Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, N.Y. 10598

Abstract

The user of interactive systems must learn a different interface for each system he uses. Furthermore the designer of such systems has limited guidelines to create good user interfaces. We describe an application interface development environment, AIDE, in which one can create and select multiple interfaces easily for a given application, and conversely one can create multiple applications with a given interface. This benefits the end-user by providing the possibility of familiar, even identical, interfaces among wide ranges of products, and this helps the designer by supporting Human Factors testing of interfaces. We formulate a model of interactive systems in which the application and interface are decoupled and the components of the interface can be changed. We also provide tools for these components and a methodology to create and select interfaces.

1. Introduction

The new user of an interactive system must learn both the functionality of the system and the procedures to invoke its functions. For example, for an editor application the user must be aware of the editor's capabilities such as to locate, delete, copy, move arbitrary text strings, and in addition he must know how to select the text strings and issue the commands. Often he can transfer his knowledge of the functionality from one system to another; rarely can he do the same for the form of the interaction process. Time-consuming for novices and experienced users alike is the need to learn both the functionality of the application and the form of the user interface. It is especially frustrating for experienced users to know what is required (e.g. to identify the third paragraph to the system and issue a delete command), to know how to do it using another software package, but not to know the specification procedure for the present package.

From the designer's point of view, creating good user interfaces is a difficult process that requires iterations of design and evaluation. However, once a system is developed, it is often too late to make major changes to the interface. Furthermore, interfaces that may be good for one user or one application may not be good for another. In response, designers may try to build flexibility into the interface, such as by using parameters or options tables, to adapt to the user and grow with his expertise. However, in general the designer cannot foresee all the desired alternatives and provide enough flexibility to tailor the interface to all users' needs.

The objective of the application interface development environment, AIDE, that we will describe is to address the above needs of both the user and the designer. AIDE accomplishes this by providing an environment in which one can create and select multiple interfaces easily for a given application, and conversely one can create multiple applications with a given interface. This benefits the designer by supporting Human Factors testing of interfaces, as well as provides the capability to tailor the interface to the user. This further benefits the user by providing the possibility of familiar, even identical, interfaces among wide ranges of products. Additionally AIDE makes it easier to develop applications, because the application designer does not have to build the interface portion of his system.

Previous work to provide assistance to the interface designer falls broadly into three areas. One area of research involves simulating systems before they are built to improve the interface design through observation and modification of a model of the system. Hanau and Lenorovitz [8] develop tools to create and present display snapshots and to vary the display based on user input and thus provide a visualization of the proposed system. Gould et al. [7] simulate the functions of a listening typewriter with a behind-the-scenes human typist, Kelley [12] simulates a calendar application with

the help of a human interpreter, and Good et al. [6] simulate an electronic mail application where a human operator augments the system's capabilities. These experiments demonstrate the feasibility of simulating the application and capturing user input to shape the interface design early in the design phase.

A second area of research to improve the design of user interfaces is work that describes formally the structure of human-computer interaction. Moran [14], Foley and Van Dam [3] and Nielsen [15] describe the user's model of an interactive system in terms of a layered hierarchy from a task level through levels of semantics and syntax to a physical level. Each level reveals more details of the form to accomplish the function of the higher levels. Moran [14] suggests designing and evaluating interfaces in a top-down approach by level with the important concept of being able to choose among varied implementations at each level. Foley and Van Dam [3] apply this framework to interactive graphics where they view a comparable hierarchy for both the user input and the computer output. Nielsen [15] uses the hierarchy model to suggest design guidelines. Reisner [17] develops a grammar based on Backus-Naur Form to describe user actions and applies this representation to analyze alternative interface designs in terms of ease of use. Lindquist [13] formulates a language that describes the dialogue structure between user and computer and then demonstrates its applicability to evaluate interfaces. Foley and Wallace [4] and Jacob [11] apply state transition diagrams to specify user interfaces for interactive systems. Of further interest, the Military Message System described in [11] is designed so that the user interface is decoupled in this application and can be described separately from the application.

A third area of research is in developing tools to generate user interfaces. Heffler [10] describes a system called MCIS which allows the designer to create menu interfaces interactively, and Buxton et al. [1] present a user interface management system, MENULAY, which gives the designer the ability to create menu-driven interfaces graphically. Olsen and Dempsey [16] generate graphical user interfaces directly from a grammar describing the interface. The Cousin System [9] converts interface specifications established by filling in forms into a uniform user interface. Ciccarelli [2] formulates a model of the user interface in which the user manipulates a presentation data base consisting of the displayed information which in turn revises a data base of the application. He provides tools to build, edit and display the presentation data base and to support queries and updates to the application data base.

In this paper we formulate a model of interactive systems in which the application and interface are decoupled and the components of the interface can be changed. We also provide tools for these components and a methodology to create and select interfaces. The approach taken in AIDE is three-leveled. First, we separate the application from the user interface, that is the function from the

form. In so doing, we provide sufficient information about the application to the interface to establish a communication path along which the interface can send user input to the application and receive application results. Second, we make the interface portion modular so that we can replace individual modules to produce different interfaces. Third, we provide presentation services tools for the common elements needed for interactive interfaces. Tools can call upon other tools to translate user input to the application and application output to the user. We describe each of these aspects of AIDE in sections 2, 3 and 4.

2. Separation of Application from Interface

In this section, we describe the AIDE architecture of separating the application from the user interface. First, we define the user interface portion of a system independently of an application. We then show how to provide the application information necessary to link an interface to an application. We describe the interface code and the generic flow of an application in this architecture.

We think of the application as the tasks to be performed and the interface as the form of the user interaction. To accomplish the separation, we need to identify the elements that comprise the interface portion of a system. We specify an interface by identifying the form of the user input and the system output and the interaction areas where this takes place. The form of the input consists of identifying how commands and arguments (objects) are given to the system. The form of the output consists of identifying how the system provides prompts, messages and results to the user. The "how" in the above statements requires specifying the medium, syntax and placement. An interface is then determined by the specification of these Interface Description Items:

1. Interaction Areas
    a. Form of interaction areas. This establishes whether there is windowing for instance, if it is overlapping or tiled, fixed or moveable, its size and position, behavior of cursor, etc. (Interaction areas are not restricted to visual forms and include audio.)
    b. Type of information to link to each area. The type of information identifies what can be displayed (played for audio) in each area, such as a document or a list of commands.

    The items listed below define the form of the input and output and can vary by interaction area.
2. Input
    a. Form of object identification, including media to use. This determines how the user will specify the objects, such as by pointing, circling or naming.

b. Form of command issuance, including media to use. This determines how the user will invoke a command, such as by selecting from a menu, speaking a command name, direct manipulation of an object. This includes the form of interface objects (such as icons) used to invoke commands.

c. Order of specifying command and arguments, if not dependent on specific commands of the application. This allows the interface to set up a uniform procedure for entering commands if desired and if not, it allows the procedure to vary by command or command groups.

d. Nesting of commands, if command independent. This specifies whether commands must be completed before others are invoked.

3. Output

a. Form of system response, including media to use. The form of the response could be echoing, completion of input, voice acknowledgement.

b. Form of prompts and help messages.

c. Form of application output.

We denote an interface style as the specification of the above interface description items either fully or partially. The designer thereby determines what constitutes a style. A partial specification can result from not defining all items or defining an item incompletely. The designer can create sub-styles by completing the specification of a partially-specified interface. For example, the designer may determine an interface style to use highlighting for the form of system response and a sub-style to be reverse video. Further specification of the interface, such as media used (audio commands instead of keyboard input), can be provided at the sub-style level. We will use for the illustrative examples in this paper a menu interface style in which objects are selected first from one menu and then commands from another menu. We will denote this interface style by style M, and we define style M in terms of the interface description items as follows:

1. Interaction Areas

a. Form of interaction areas - Display screen has four fixed windows: Application Display Area (0,4)-(A,B), Command Menu Area (0,0)-(A/2,4), Object Menu Area (A/2,0)-(A,4), and Prompt Area (0,B)-(A,B+1). (Note that the units here are some length measure on a coordinate system with (0,0) being the upper left corner of the display screen.) The cursor can move freely between windows with no implied scrolling capability.

b. Type of information to link to each area - The Application Display Area, Command Menu Area, Object Menu Area, and Prompt Area are linked to the application's output, commands, objects, and messages, respectively. (Information for this linkage and for the tokens to display is given by data files that will be described below.)

2. Input

   a. Form of object identification - Identify object by selecting in Object Menu.

   b. Form of command issuance - Invoke command by selecting in Command Menu.

   c. Order of specifying command and arguments - The object is selected first then the command.

   d. Nesting of commands - A command must be completed before another is invoked.

3. Output

   a. Form of system response - Selection of object and command will be highlighted in menu. Selection will be echoed in Prompt Area.

   b. Form of prompts and help messages - Messages will appear in Prompt Area , if there is a selection error.

   c. Form of application output - Application output will be in Application Display Area with the application specifying the relative placement.

It is necessary to provide application information to the interface in order to establish a link between the user specifications and the application. We identify the commands and objects of the application to the interface by a file which serves an analogous function to the schema in a data base management system. The file has two parts consisting of the Message File and the Context File. The Message File is straight-forward: it contains the prompts and help messages for the application. These can be changed without affecting the application or interface code. An identifier is associated with each message in the file and used by the interface to select the appropriate message.

The Context File contains an equivalency table that specifies the user-issuable tokens for an interface and the corresponding identifications known by the application. The table tells the interface how to translate the user's input to the commands and objects of the application. A hierarchy of commands and objects is specified with the table. A given application has a different Context File for each interface style to identify the corresponding user representation of each application command/object in that interface style, such as the input tokens in a command line style, the function key numbers in a function key style, or the icon descriptions in an icon style. In addition, any command-dependent exceptions to the interface style are specified. The Context File grammar is not defined fully yet; however the following is an example to show the elements of a Context File for a

forecasting application using interface style M. Each line of the Context File that defines a command or object has the syntax of interface token followed by an arbitrary number of application identifiers. We specify the hierarchy of the application commands and objects for the given interface by indentation. Exceptions to the interface style are indicated by an "&" preceding the interface identifier, which is then followed by a list of the exceptions to the style rules.

```
window 0
  commands
    sum     0
    update  4   2
    project  1
    display
      window 0.0
          numerically  2
          graphically  5
    quit    3
    & quit   !O   >V
    & update  >D

window 1
  objects
    sales   s1   2
    cost1   c1   2
    cost2   c2   2
```

Figure 1. Context File for Forecasting Application, Style M.

In this example, the Context File supplies the interface with the names of the menu elements to display in two windows and their corresponding identification to the application. The Command Menu Area and the Object Menu Area of style M are thereby linked to the elements of window 0 and window 1, respectively. If the user selects "project" from the Command Menu Area, the interface

would send the message 1 to the application. This file also gives an example of the command hierarchy. The selection of "display" in window 0 invokes sub-window 0.0 to allow the selection of the form of the display. Exceptions to the interface style are specified in the Context File. The above example provides for the following command-specific information: the "quit" command selection does not require an object selection; the "update" command requires data; the interface will ask for verification of the "quit" command before sending the message to the application. A command or object selection can send multiple messages. For example if the object "sales" is selected in window 1, the application would receive a message telling it that the object identified by s1 was selected and to invoke the command given by command identifier 2 to display it numerically. Similarly, the selection of "update" will send a message to the application with command identifier 4 to update and command identifier 2 to display the result of the update.

The application programmer can provide in the Context File as much or as little information about the application's data structure as he desires. The more information provided to the interface by the Context File, the more the interface can do for the application. At one extreme, if no object information is provided, all the interface may do is identify the relative position pointed to by the user in a window. If the application's data structure is provided to the interface, the interface can tell the application what object has been selected.

The interface description items listed on pages 3 and 4 are mapped to application-independent code. This code establishes an interface and uses the data files (Context and Message) to link this interface to a specific application. In addition, parameters such as window position and size can be supplied to the interface routines by an interface data file for the selected style. We call the application-independent code the Interface Routines, which consist of a Setup, Prompt, and Display routine. The Setup routine creates the interaction areas and links each area to the appropriate data based on the Context File. It uses the Context File to establish the correspondence between the interface tokens and the application. The Prompt routine interacts with the user to assemble user input. It translates the user input to the application based on the interface/application relationships established by the Setup routine. The Display routine provides the results of the application execution to the user. Interface description items 1a and 1b determine the Setup routine; items 2a, 2b, 2c, 2d, 3a, and 3b determine the Prompt routine; item 3c determines the Display routine.

There are multiple instances of each interface routine. A specific instance of each of the three routines, Setup, Prompt and Display, is selected based on the interface style desired. The following is an example of the interface routines for style M that requires object then command order specification. These interface routines use the Context File shown in Figure 1.

```
Setup Code Style M
   Create Application Display Area
   Display Data in Application Display Area
   Create Command Area (Context File)
   Display Data in Command Area
   Create Object Area (Context File)
   Display Data in Object Area
   Create Prompt Area (Message File)
   Display Data in Prompt Area
   return

Prompt Code Style M
   Accept Object Identification from object area
   If error,
   {  Display error__1 (Message File) in prompt area
      goto Accept Object }
   Respond Object Identification
   Accept Command Identification from command area
   If error, Display error__2 (Message File), goto Accept Command
   Respond Command Identification
   If Command requires data (specified by Context File)
   {  Accept Data Identification
      If error, Display error__3 (Message File), goto Accept Data
      Respond Data Identification }
   return

Display Code Style M
   Display Data in application display area
   return
```

Figure 2. Interface Routines for Style M

The Prompt Code in this example includes a conditional statement that depends on the command information in the Context File. There can also be conditional statements based on object type information and interaction area that can be specified in the Context File.

Once we separate the above code that handles user interaction from the application code, all applications consist of objects, functions to manipulate the objects, and the data files (Context File and Message File) which describe the commands, objects, and messages to the interface. The application flow consists of a call to the Setup routine, then iterations of call Prompt, execute selected command, call Display.

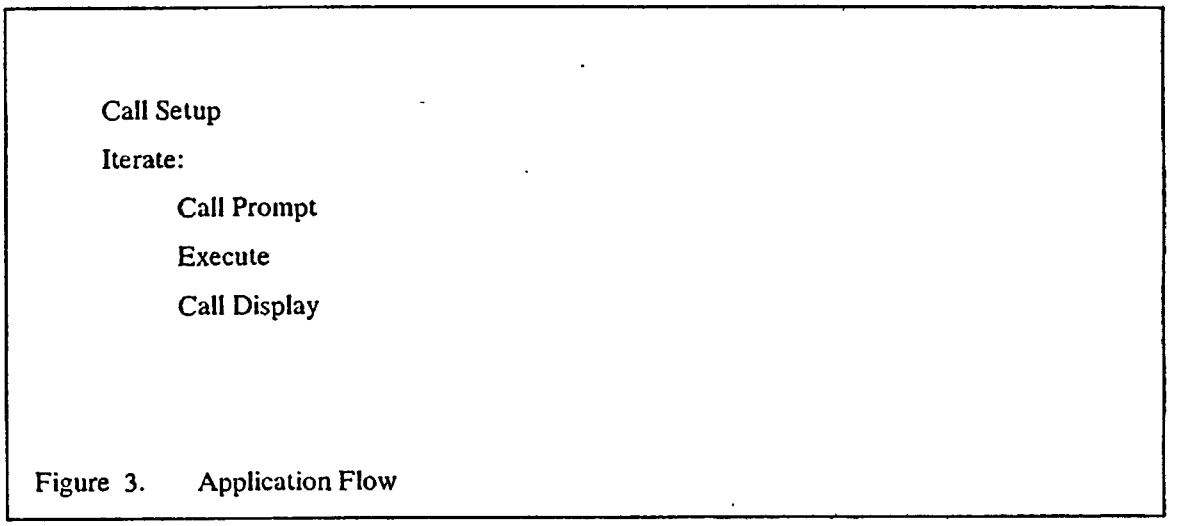

Figure 3. Application Flow

Applications participate in this environment simply by including calls to the interface routines Setup, Prompt and Display and providing a Context and Message File. Participating applications contain no interface code; they receive their input from the Prompt routine and pass their output to the Display routine.

The Setup routine uses the Context File to establish the desired interaction areas and link the appropriate data. The Prompt routine prompts (using the Message File) the user to provide input, assembles the resulting user input by the parsing mechanism established by the Setup routine, provides an acknowledgement to the user or request to edit errors (using the Message File) and then sends the assembled command to the application. The application will include a case statement which executes the required application functions based on the assembled command. Then the application sends the result to the Display Routine which provides it to the user.

The Prompt routine can call upon Setup, if the user has issued a request to change the interface style.

3. Modularity

In this section, we describe the flexibility that the AIDE architecture provides to change interfaces.

One level of modularity is attained by the ability to select the appropriate interface routines based on style. Changing styles can then be accomplished by selecting a different instance of the interface routines.

As seen in Figure 2, the statements that comprise the interface routines are functional, such as Create Display Area, Accept Input or Display Output. The specific method to perform the Create, Accept or Display and the form and medium of the Display Area, Input and Output are determined by pointers to selectable code. The interface routines contain functional statements that are later expanded into code. The actual code is set up in modules that are selected by pointers which provide the second level of modularity. Modules can interface with other modules to form a communication path between the user and the application. Any module or group of modules along the communication path can be replaced as long as the message interface at that point is preserved. As a result, the form of the message may change, provided that its meaning is preserved. A module may add meaning to the message, for example by correcting misspellings through a spelling checker or by interpreting voice input by a speech recognition module.

The modules along the communication path interpret and modify the message and send it on. The user's input is a message that is passed along by the modules comprising the interface to the application. We call these messages incoming messages. The application then produces a result based on the incoming message and sends the outgoing message to the user through the interface modules. The application is analogous to an object in object programming [5]. It receives a message, selects the appropriate method to accomplish the task, and sends out a message.

There are cases when instead of passing the message on, a module will send an incoming message back toward the user or an outgoing message back toward the application. This may occur with an incoming message, because:

1. An error is detected (such as unrecognized input, perhaps a misspelled command or unrecognized audio instruction),

2. Enough information is available to complete the request at the interface level (such as text edit), or
3. A preset dialogue loop has been established (such as the interface style may require a verification query when the user selects "quit").

Similarly with an outgoing message from the application, a module will not send a request on to the end-user, if the interface routine can complete the request.

As an implementation method, a Code Dictionary provides the mechanism to select for each functional statement in the interface routines the desired code module or modules. An entry in the Code Dictionary consists of a style key and sub-style key (if necessary), the functional statement of the interface routine, pointers to the appropriate code modules, and parameters to input to modules. A given entry can have several pointers to select a path of modules. Furthermore, the Code Dictionary is multi-leveled, so that an entry in the Code Dictionary can point to another entry to incorporate the other entry's path.

4. Presentation Services Tools

The interfaces for interactive systems require many of the same modules to display windows, point to objects, edit input. Since many interactive interfaces have common elements, a tool kit approach is appropriate for these common elements. In this section, we define the organization of the presentation services tools.

The main components of the tool kit are a window manager, dialogue managers, input/output interaction tools (handwriting recognition, speech digitization, speech recognition, speech synthesis), graphics package, multi-media editor. These form the modules pointed to by the Code Dictionary. Thus, when an interface style calls for a visual display area, the Code Dictionary expands the Create functional statement in Setup by pointing to the window manager tool and providing it with the appropriate parameters. For an interface style receiving input from handwritten commands, the Code Dictionary expands the Accept functional statement in Prompt by pointing to the command line dialog manager which in turn calls the handwriting recognition tool.

Figure 4 shows a diagram of the AIDE framework. The shaded boxes identify application-dependent code and data; the white boxes signify application-independent code. The application calls upon the interface routines whose functional statements are expanded into code by presentation services tools that are selected by the Code Dictionary. The instance of the interface routines and the specific tools selected depend on the interface style.

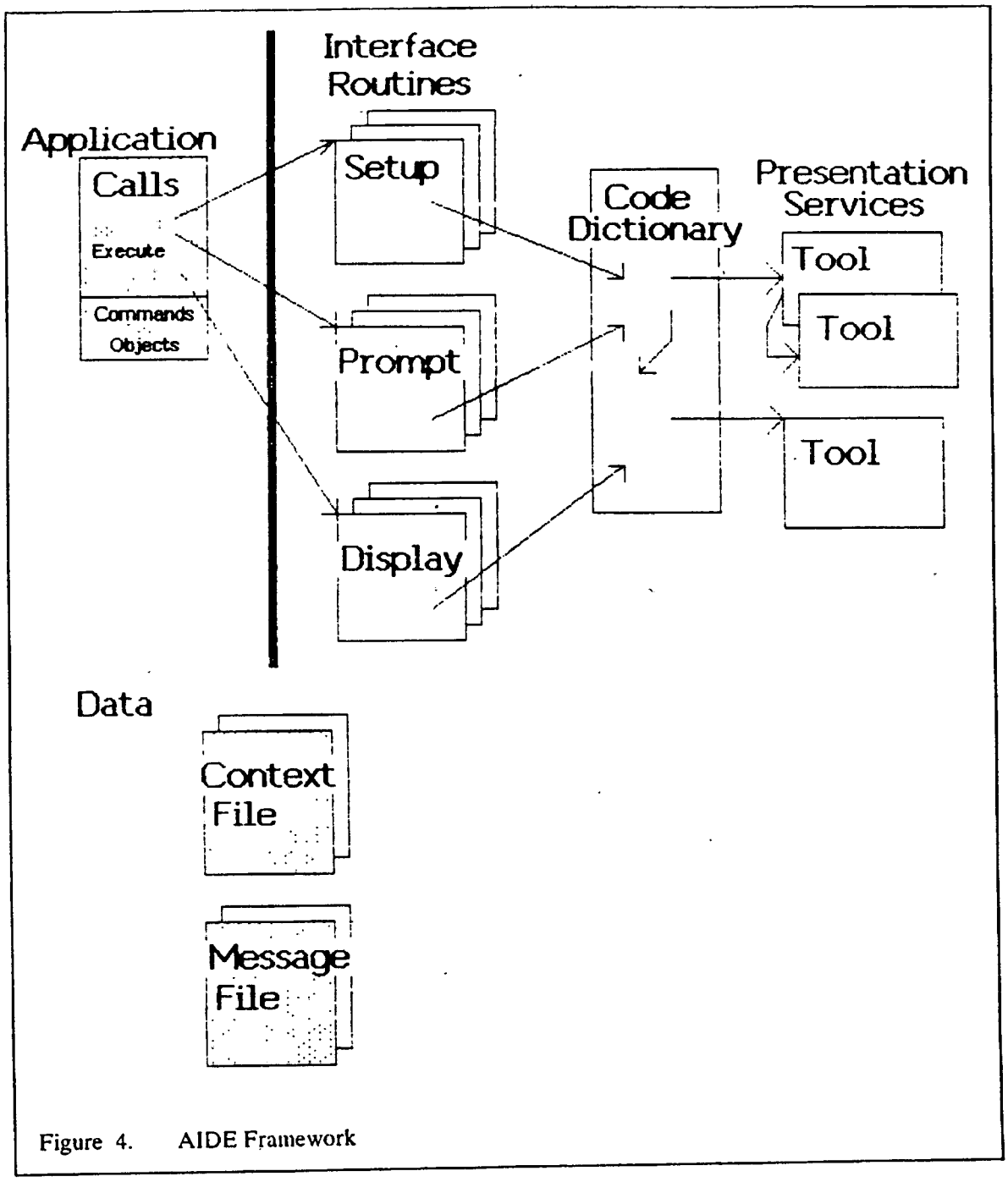

Figure 4. AIDE Framework

The tool kit is extensible to include new tools as developed, and there is also a certain amount of freedom in defining the tool kit. The tool kit is modular in that tools can be linked to other tools, and it is multi-leveled in that advanced tools call upon basic tools to accomplish portions of their task. An example of the modularity is the menu dialogue manager calling upon a speech recognition tool for input. For another interface style, it may call upon a keyboard tool or a handwriting recognition tool. An example of the hierarchy of tools is the menu dialogue manager using the window manager to set up the menus.

We include a brief description of the main elements of the tool kit. The window manager maintains a data base of rectangular regions, associating with each an identification, position, priority and pointers to data. It provides an interface to data base management services to create and manipulate a multi-window display of possibly overlapping documents.

The dialogue managers provide pre-programmed interactions with the end-user. The input portion of these interactions consists of command and argument specifications from the end-user, and the output portion consists of prompts and responses from the dialogue managers. The dialogue managers use the information supplied by the Context File to identify the input and translate it to the application. There is a dialogue manager tool or tools for each kind of interaction area, such as panels (input fields, soft keys, menus), command line, icons.

The input/output interaction tools provide the translation from different media to coded information and vice versa. Examples of input/output interaction tools are handwriting recognition, speech digitization, speech recognition, speech synthesis.

The graphics tool provides the functions for interactive graphics. These include a number of primitive geometric objects which can be combined to form other objects. It also contains functions to manipulate these objects as well as any others provided to the system (for example, by drawing).

The editor tool provides a set of edit commands that can be applied to multi-media documents. Editing can take place in any interaction area (application display areas, command line, menu, etc.) and is applied to both application objects and interface objects in a uniform way.

The tools must be designed so that they can interface with each other to modify the form of the message along the communication path (see Figure 5). First, tools create the interaction areas and establish the application/interface relationships from the Context File as part of the Setup interface routine. Tools then use these relationships to translate messages from the user to the application. These tools comprise the Prompt interface routine for a given interface style. The translation converts the message from user syntax to application syntax, possibly amplifying the meaning along the way. The application then sends back a message through a series of tools that form the Display interface routine to translate (and possibly amplify) the message to the user. As stated above, any tool along the path may send back a message to its sender, if the tool detects an error or has enough information to respond itself to the message.

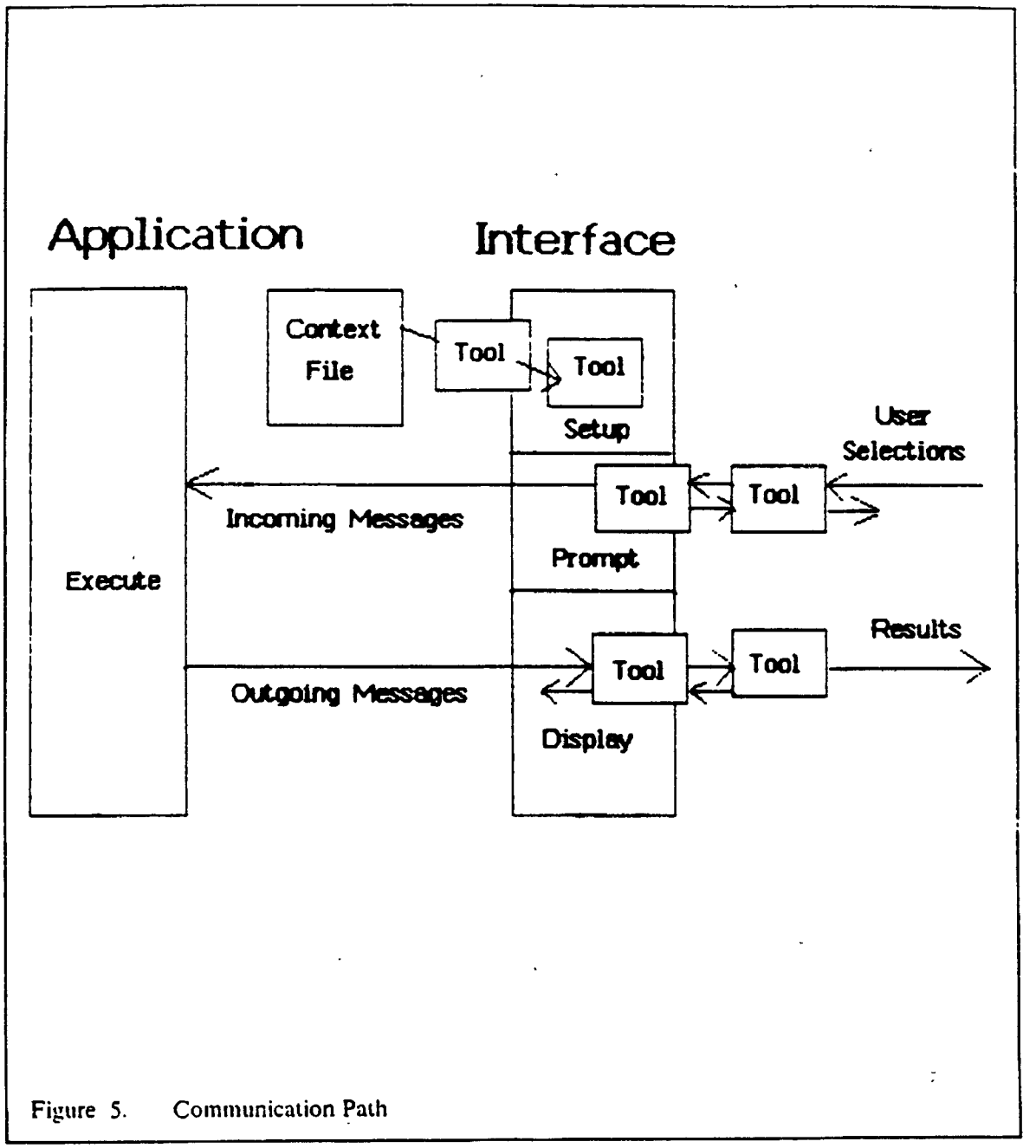

Figure 5. Communication Path

5. *Creating and Changing Interfaces and Applications*

The AIDE framework decouples the application from the interface. In this section, we discuss how to create an interface in a given style using the AIDE framework and show how to change the interface and the application independently.

AIDE allows the interface designer to experiment with interfaces for a given application by providing the capability to select from existing interface styles as well as to create new interface

styles. In both cases, the Context File must be constructed for that interface style and specific application.

To create a new interface style, the designer specifies the interface description items shown on pages 3 and 4 to describe the Interaction Areas, Input form and Output form. A generator program would then create the interface routines from these specifications and provide the pointers and parameters in the Code Dictionary to select the appropriate code modules and provide the necessary data. In our example, the menu-style interface M would be generated from the specification of the interface description items for style M on pages 4 and 5. The generator program would create the interface routines of Figure 2 and the entries in the Code Dictionary to select the presentation services tools. Sub-styles can be created by specifying in the interface description items, for example, the form of highlighting to be reverse video or the form of menu selection to be voice input. Similarly, the designer can create a direct icon manipulation style or a command line style by completing the interface description items for those styles.

To choose from existing interface styles, the designer selects a style key to identify the appropriate Setup, Prompt, and Display routines, and the routines' functional statements are then expanded into code by the Code Dictionary using the style and sub-style keys provided. For example, providing the key for style M would result in the selection of the interface routines in Figure 2. The functional statements of these routines, such as "Create Application Display Area" and "Accept Object Identification", would be expanded into code through Code Dictionary pointers, in this case to the window manager and to the menu dialogue manager, respectively. Sub-style keys allow the designer to fine-tune the interface by specifying the tools to be selected by the Code Dictionary. As a further example, if the designer wanted to change interfaces to a command line style C, in which the end-user enters a command followed by an object and possibly additional data in a command area, the designer would provide a key to select a Prompt routine such as the following:

```
Prompt Code Style C
   Accept Command Identification from command area
   If error, Display error__2 (Message File), goto Accept Command
   Respond Command Identification
   Accept Object Identification from command area
   If error, Display error__1 (Message File), goto Accept Object
   Respond Object Identification
   If Command requires data (specified by Context File)
   {  Accept Data Identification from command area
      If error, Display error__3 (Message File), goto Accept Data
      Respond Data Identification }
   return
```

Depending on the sub-style selected, the command area could be the keyboard, audio, tablet.

The interface style or sub-style can be changed at one level by replacing a presentation tool module (handwriting recognition for audio recognition), at a higher level by replacing the interface routines (direct icon manipulation for menu selection). Thus what appears to be a major change to the user is accomplished by a change of pointers to the presentation services modules or by a change of pointers to the interface routines themselves and with no change to the application.

On the other hand, we can change the application and keep the same interface. The designer accomplishes this by modifying the Context File associated with that interface style to include the new application's tokens and by providing the new application's messages in the Message File. As a simplified example, we change applications from the forecasting application using interface style M to a mail application in style M by modifying the Context File of Figure 1 to the Context File shown in Figure 6.

```
window 0
  commands
    read    0
    write   4   0
    send    2
    delete  1
    quit    3
    & quit   !O   >V
    & write  >D
    & send   >D

window 1
  objects
    note1   n1   0
    note2   n2   0
```

Figure 6. Context File for Mail Application, Style M.

The application programmer would not have to make any change to his program or to the interface routines to inherit the interface.

6. Conclusions and Future Work

We have shown how the AIDE framework provides the ability to change user interfaces and applications independently of each other to benefit both the end-user and the designer. We described how to generate the user interface automatically, either directly from the specification of the interface description items or by the selection of pre-established routines by a style key. We think that this approach will strengthen the interface design procedure and improve the usability of interactive systems.

Future work on AIDE will involve developing grammars to 1) specify the interface description items to the program that generates the interface code, 2) describe the elements of the Context File to the interface routines, and 3) construct the messages between the application and the interface routines.

Acknowledgement

The authors would like to thank S. J. Boies, J. M. Carroll, C. L. Cesar, G. B. Leeman, R. L. Mack, M. A. Martin, P. Reisner, and J. R. Rhyne for their helpful comments and suggestions; G. B. Leeman for suggesting we investigate user interfaces in the first place and for his many thoughtful comments which helped improve this report; J. M. Carroll for providing insight into modelling human-computer dialogue; and M. A. Martin for valuable discussions on object-oriented programming.

REFERENCES

1. Buxton, W., Lamb, M.R., Sherman, D. and Smith, K.C. Towards a Comprehensive User Interface Management System. Computer Graphics, 17, 3, July 1983, 35-42.

2. Ciccarelli, E.C. Presentation Based User Interfaces. MIT/AI TR 794, Massachusetts Institute of Technology, Aug. 1984.

3. Foley, J.D. and Van Dam, A. Fundamentals of Interactive Computer Graphics. Addison-Wesley, Reading, MA, 1982.

4. Foley, J.D. and Wallace, V.L. The Art of Natural Graphic Man-Machine Conversation. Proceedings of the IEEE, 62, 1974, 462-471.

5. Goldberg, A. and Robson, D. Smalltalk-80 The Language and its Implementation. Addison-Wesley, Reading, MA, 1983.

6. Good, M.D., Whiteside, J.A., Wixon, D.R., and Jones, S.J. Building a User-Derived Interface. Communications of the ACM, 27, 10, Oct. 1984, 1032-1043.

7. Gould, J.D., Conti, J. and Hovanyecz, T. Composing Letters with a Simulated Listening Typewriter. Communications of the ACM, 26, 4, Apr. 1983, 295-308.

8. Hanau, P.R. and Lenorovitz, D.R. Prototyping and Simulation Tools for User/Computer Dialogue Design. Computer Graphics. 14, 3, July 1980, 271-278.

9. Hayes, P. Personal Communication, Mar. 11, 1985.

10. Heffler, M.J. Description of a Menu Creation and Interpretation System. Software - Practice and Experience, 12, 1982, 269-281.

11. Jacob, R.J.K. Using Formal Specifications in the Design of a Human-Computer Interface. Communications of the ACM, 26, 4, Apr. 1983, 259-264.

12. Kelley, J.F. An Iterative Design Methodology for User-Friendly Natural Language Office Information Applications. ACM Transactions Office Information Systems, 2, 1, Jan. 1984, 26-41.

13. Lindquist, T.E. Assessing the Usability of Human-Computer Interfaces. IEEE Software, 2, 1, Jan. 1985, 74-82.

14. Moran, T.P. The Command Language Grammar: a representation for the user interface of interactive computer systems. Int. J. Man-Machine Studies, 15, 1981, 3-50.

15. Nielsen, J. A Virtual Protocol Model for Computer-Human Interaction. DAIMI PB-178, Aarhus University, Sep. 1984.

16. Olsen, D.R. and Dempsey, E.P. SYNGRAPH: A Graphical User Interface Generator. Computer Graphics, 17, 3, July 1983, 43-50.

17. Reisner, P. Formal Grammar and Human Factors Design of an Interactive Graphics System. IEEE Trans. on Software Engineering, SE-7, 2, Mar. 1981, 229-240.