1

Selected Ingredients in End-User Programming

Moshe M. Zloof

Principal Architect, Hewlett-Packard Laboratories

1501 Page Mill Road, Palo Alto, CA 94304 USA

Tel: (650) 857-7485 E-mail: zloof@hpl.hp.com

Abstract

In the area of human computer interaction, over the last twenty years, we have witnessed considerable progress in an ever-increasing bandwidth from the computer to the user. Application screens evolved from static text only screens to interactive GUI screens. These screens contain numerous graphical element or "widgets", supporting multiple data types, such as text, voice, image, and video. The widgets can range from simple ones like a combo box or slider to more complicated OCX's such as interactive graphs or maps.

On the other hand, the tools to program this application are still in the domain of programmers. Although there has been much progress in various RAD tools, visual language and 4GL to improve ease of use, they still mostly target programmers. We believe that in order to allow end-users to develop their own advanced UI applications, it is necessary to create higher-level application abstractions or 'algebra' for stating the application in a declarative manner. This can be compared to the relational algebra operators in the data base area. They were created as abstractions for data base queries, enabling end users to express their own queries in a declarative manner. In doing so, bugs are minimized and program modifications and maintenance becomes trivial. In this paper, we will motivate the reader to see the need for these abstractions and classify them into categories, emphasizing areas ripe for further research.

Keywords

WYSIWYG Programming, Application Abstractions, Declarative Programming

INTRODUCTION

Over the last twenty years, we have witnessed considerable progress in increasing the computer-to-human and human-to-computer bandwidth. Business applications evolved from static screens, with primarily text and tabular records, to sophisticated interactive GUI screens. These screens containing numerous graphical data elements or "widgets" support multiple data types such as text, voice, image, or video. The widgets range from simple ones like a field, a combo-box or a slider, to more complicated OCX's such as graphs, and maps; to even new objects that use novel visualization techniques like glyphs or other metaphors to depict dynamic statistical data fall under the category of widgets.

Unfortunately, the tools and languages available to create such applications are still in the domain of professional programmers. Because of the general-purpose nature of many RAD tools such as VisualBasic (1), JavaScript, and other 4GL and visual languages (2), (3), very few cater to real end users. Most of them improved the textual linear programming by adding tool bars of various widgets that the user can drop on the screen, Fig. 1. Nonetheless, the inter-widget events that take place when one interacts with these widgets, have to be coded by professional programmers, the back of Fig. 1.

Current State of RAD Programming

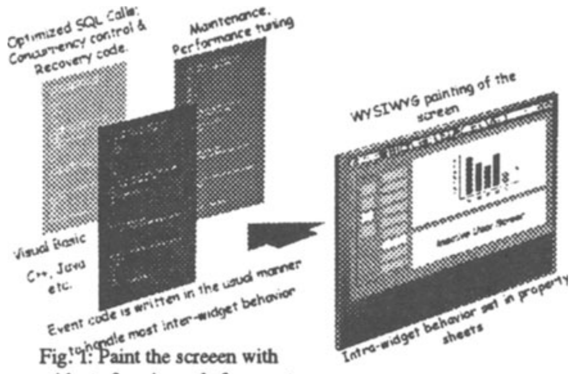

Fig. 1: Paint the screeen with widgets & write code for events

Granted, some improvements have been made by introducing various WIZARDS to improve the programmer's task, but in almost all cases, programmers still have to

delve into the code arena. On the other hand, systems targeted to end-users are quite specialized and limited to a particular Application Domain such as FABRIK (4) or LabVIEW (5). Perhaps the only programming environment that can really be touted as an end user tool is the spreadsheet. It is in a way, very general purpose because it can be applied to a myriad of domains. Yet it is limited to the structure of the spreadsheet grid. We believe there is a class of applications in the area of advanced GUI, which can go beyond the power of spreadsheets and still remain in the realm of end-users. It has applicability domains similar to spreadsheets. In order for end users to create their own advanced GUI applications, it is necessary to create higher-level application abstractions or 'algebra' for stating the behavior of the application in a declarative manner. This is alien to the creation of data base query abstractions so end
users can express queries in a declarative manner.

APPLICATION CHARACTERISTICS

Before describing the abstractions, let us elaborate on the characteristics of these applications. Ben Shneiderman of the University of Maryland, in his *Data Type Taxonomy of Information Visualizations,* (6) argues that a good advanced graphical user interface for multidimensional data visualization must allow the user to 1) overview the data, 2) zoom, 3) filter, 4) get details-on-demand 5) relate: view relationships among items, 6) keep history, and 7) issue parameterized queries. Normally, these functionalitys have to be programmed and hardwired for each application. What we are after is a set of algebra or abstractions that allow an end user to build applications of the above power declaratively. These applications have the following characteristics:

1. Allow the creation of multiple interactive GUI screens with multiple data and control widgets.
2. Allow browsing, navigation and updating of the data.
3. Data sources can come from such multiple sources as databases, spreadsheets, or instruments.
4. Can be either deployed as a stand-alone application, client, server, Intranet or Internet.

APPLICATION ABSTRACTIONS

Abstractions in Programming Languages.

The concept of abstractions in programming language is not new. Even early programming languages had them such as data types, functions, etc. Later in object-oriented languages, abstractions such as inheritance and encapsulation were introduced to facilitate the programmer's task to write and maintain repetitive code.

Abstractions in DBMS:

Prior to relational data base management systems (DBMS), the task to program a data base application was quite tedious since it dealt with every aspect of the process, writing low-level code, specifying the selection path and code to optimize the program code for recovery. The advent of DBMS's enabled the programmers to write declarative programs (SQL, QBE) and basically state queries in a declarative 'what-you-want' manner. The rest, such as choosing selection paths, optimization, concurrency and recovery, was relegated to the DBMS, thus eliminating the necessity of programming it for each application. However, in order to express a program in a declarative manner, it was necessary to create many data base abstractions. Examples of such abstractions are: relational algebra operators; SELECTION, JOIN, and PROJECTION and keys. Prior to introducing the JOIN operator, for example, programmers had to communicate with each other and the end-user using the semantics of the JOIN. By explicitly introducing the JOIN abstraction and defining its formal semantics, a common ground was created for programmers and end-users to communicate in a declarative manner when using one of the operators as part of the query. Other abstractions are concepts such as key attributes (i.e., cannot have duplicates), foreign key attributes, groupings, and abstractions for modifying the database (such as cascade delete, or restrict delete).

GUI Application Abstractions

As mentioned earlier, in order to facilitate the tedious programming effort to build an advanced GUI application, we need to create abstractions at the application level. The nature of these abstractions must be as follows:

1. They have to be declarative – i.e., stating 'what-you-want' declaratively.
2. They must make sense to end-users, i.e., you can easily explain the semantics of the abstractions and the user can relate to them.
3. Translate into considerable amount of code otherwise it does not make sense to introduce a new concept that can otherwise be expressed with a few lines of code.

As we shall see, normally these abstractions take advantage of constraints imposed by the underlying systems or the data sources.

AREAS OF ABSTRACTIONS

In creating abstractions for the class of applications mentioned above, we identified five areas within which abstractions are necessary if we want to eliminate the drudgery of writing code. If you take them collectively, they constitute an 'algebra' by which end-users can create these advanced user interface applications. In this section, we are going to list these areas and motivate the need for these abstractions. But we will address only one in detail – namely the Interaction Abstractions.. The description of these abstractions is quite informal and it will be motivated by examples of why we believe it.

Rationale for the Need of the Abstractions

When there is a large body of multimedia data to be visualized or 'rendered' on the normally limited screen or multiple screens, it is obvious that one cannot present the entire data in one shot. Consequently, various graphical elements "widgets" like combo boxes, outliners, and sliders were invented to facilitate the presentation by displaying at any one time, a subset of the data. The combo box, for example, selectively picks a data value from a list, which may have an effect on data, presented in other widgets. Picking, for example, a customer name from a combo will cause the list of ordered products to change for that customer. Moving a knob on a slider can dynamically filter and eliminate data presented in other widgets (as in dynamic queries)(7). The outliner, on the other hand, opens and closes various directories to save real estate space and also to prevent clutter if all directions were opened at the same time. Traditional programming using a state of the art GUI builder would require the following steps:

1. Place the widgets on the screen; i.e., paint the screen.
2. Program each event for the widgets, wherein the semantics of the interactions are specified. These include changing data in other widgets and restricting/facilitating the choices of interactions in widgets, Fig. 1.

This type of event programming deals with not only the inter-widget relationships but also with the idiosyncrasies of the individual widgets and the windowing environment. Thus, the resulting code is fairly complex. Further, the semantics of interacting with a widget can potentially affect the data in many other widgets. As a result, the programming complexity grows non-linearly with the number of widgets on the screen. Users/customers tell the programmers how each screen must behave and after it is programmed; the user may tune it by new requirements, which may require additional programming or modification of the existing programs. Furthermore, any future extensions become very tedious because the programmers who wrote the 'initial code' may not be with the company anymore and very few programmers like to delve into other people's code. And that is why maintenance of these applications cost s more than the initial development. If one can analyze and classify the variety of widgets and the interactions that one wants and create a rich set of interaction abstractions that cover 80%- 90% of what people normally need and want, then it becomes relatively easy to specify in a declarative manner these interactions without the aid of a programmer. Then it becomes the system's responsibility to translate it into code. Thus, by trial and error, one can get to the optimal running program. Changing the program becomes like editing a document because all the user has to do is to substitute one abstraction for another. Thus we need abstractions in the areas of abstracting for graphical data elements (widgets), inter-widget interactions, display abstractions, logic abstractions and data source abstractions.

Area 1: Abstractions for Graphical Data Elements (widgets)

In our definitions, widgets can range from simple graphical elements like a field, a combo-box, a slider to multidimensional widgets like a table, a graph cross-tabs to more specialized OCX's like interactive maps, glyphs, timeliners, etc. Without creating abstractions for these widgets, each has to be programmed independently and any import of a new widget into the repertoire of widgets becomes quite tedious. These widgets can be classified into categories through a set of behavioral properties: 1) can the widget represent a set or singleton, 2) can it represent duplicates, 3) can it represent instances or ranges of data and, 4) can you select a value or multi-value. Once these properties are identified, not only can the intra-widget behavior be characterized but also the inter-widget behavior can be implied. For example, if a set of values is rendered in a text-box, which usually can only represent a singleton value, then the system must provide some means to navigate to other values in the data such as back/next buttons. Further, extending the system to include new widgets can be made semi-automatic using these widget properties. As a result of this widget categorization, a GUI program has been defined to have appropriate meaning for all possible combinations of properties. In other words, the semantics of the program is not based on the individual idiosyncrasies of any widget such as text-box or radio button but of some generic properties. We conclude from this observation that any widget in a given screen can be replaced by any other widget and the "meaning" of the new program can be appropriately defined; i.e., the system will manage the intra-widget and inter-widget behavior appropriately. Furthermore, if we add some intelligence to the widgets to automatically scale themselves according to the cardinality of the data, it will save the designer the job of having to specify it.

Area 2: Inter-widget Abstractions

As mentioned earlier, when as entire screen is populated with various widgets that are linked to data sources, each interaction with a widget may affect the data in other widgets. Furthermore, it is possible that the user is interacting with a pair of widgets at a time such as dragging some elements from one widget and dragging it on another widget. Here we give the rationale for creating enough abstractions so that inter-widget event programming will not be needed for 90% of the cases.

Abstractions for a single widget interaction

1. Nested levels hierarchy abstraction

In order to make the interaction with the screen widgets more manageable, inter-widgets effects in any rendering are typically localized by some grouping of widgets that are either visually or semantically obvious to the user. Sub-forms and two-level forms are examples of this type of grouping. Such groups are necessary because if every widget can potentially affect every other widget on the screen then the rendering is likely to become incomprehensible. But groups of widgets can affect other groups, which means that the same incomprehensibility problem for widget can also affect the groups, if there are lots of groups. In order to avoid this

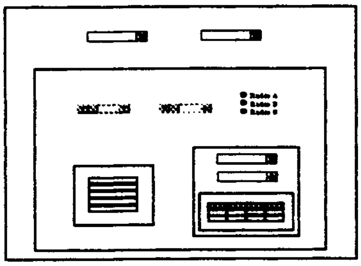
Fig. 2: Nested Levels

recurrent problem, groups are typically hierarchically ordered in most renderings and groups can only affect the groups below and not vice versa. Such a hierarchical decomposition of the set of widgets on the screen is an assumption widely used in rendering applications. We make the same assumption and term each group in the hierarchy a *LEVEL* (Fig. 2). The rendering semantics can be recursively defined based on the hierarchy and the semantics of a single level. Note that specifying the levels can be done graphically or by declaring the parent of each widget. A selection in multiple widgets in a level constitutes a conjunct filter to the levels below. In this way, by systematically going down the hierarchy and selecting data instances from various widgets, one can zoom at the desired data in the lower levels. This is the technique used in dynamic query interfaces. *Note that if the designer drags a widget from one level to another, the entire underlying code changes but the designer is not aware of this because he/she is using the level abstraction.*

2. Inter-widget Abstractions within a Single Level

We have grouped the widgets by levels, where a selection in multiple widgets in one level constitute a conjunct filter to the data in the levels below. The question is how does a widget affect other widgets in the same level? After careful analysis, we identified three categories that a designer may want to use to affect the inter-widget behavior depending on the requirements of the application. We demonstrate this by an example. In Fig. 3, we have three screens all of which have two levels but are part of three different applications. In the first contact application we have two widgets, Subject and Date. This level category is called SYNCHRONIZED because when a subject is picked, the Date value changes to synchronize it with the database record. Similarly, if the user picks a Date value first, the subject changes to Synchronize with the database. In the real estate application below, a different behavior is required. This level category is called ANCHORED. When you pick a price for a house, you don't want the number of bedrooms to change. In other words, these selections are independent of each other and choice in one has no effect on the others. The third category found in the Video application on the right is called REDUCED. This category makes sense when the volume of data is large. If you pick a "Action" under category and then click on "Actors", you expect the set of actors to be reduced to only those appropriate for action movies. Similarly, if you start with "Actors" first, by picking a particular actor and click on category, the system should reduce the set to only categories of that actor. At first glance, it may seem that the two widgets could have been placed in two different levels but then

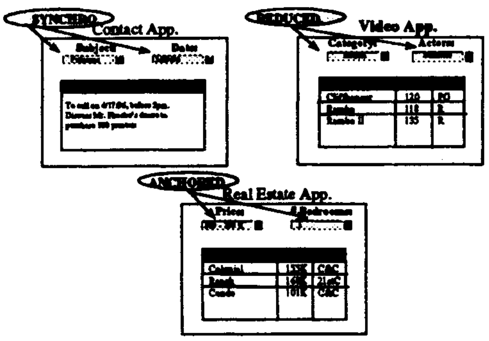

Fig. 3: Inter-widget abstractions in a single level

you are constrained to filter from the top level. Whereas what we have here is like a dynamic hierarchy, i.e., you can start from any widget. It can be argued that these three categories are comprehensive for most renderings. Thus, the inter-widget effects of any interaction can be expressed by declaring the category of the level. Note that a category of any level can be changed to another and the net effect is to give a new behavior amongst the widgets in the level. This is true irrespective of the set of widgets in that level.

Since these abstractions are not available in RAD programming, the programmer has to write code for every event. Specifically, in the example of REDUCABLE level, if you have ten widgets in a level and the designer adds an eleventh one, the code that was written for those ten widgets has to be changed to accommodate the introduction of the eleventh widget. *Whereas with REDUCABLE abstraction, the system produces the appropriate code.*

Abstractions of Interactions through a Pair of Widgets

As advanced direct manipulation applications evolve, user interactions are becoming more sophisticated. One of these is the "drag-and-drop" feature as in (8) and (9). The user can identify data elements from one widget and drag and drop it on another widget to achieve a desired task. For example, the user identifies a couple of names on a name list, drags it to a different unrelated list to see whether any of the names match the names on the second list. In another example, one can drag an item from a list of items and drop it in an order form for the purpose of ordering that item. The introduction of the web has made the users familiar with the concept of navigation through URL's. Here we can characterize these interactions as abstractions, where the user navigates a set of values and drops them on another data widget with a different set of values.

1. Relational Algebra Revisited for Navigation

When relational algebra operators were introduced, they were normally used to write a query on the schema. Even in QBE, the user expressed a query program by entering expressions directly into the schemas of the database. In today's GUI applications, especially those that are deployed on the web, we cannot expect an Internet user to write a QBE query, let alone an SQL statement, even though he/she is comfortable navigating through URL's. Here we propose to revisit all the relational algebra operators for the purpose of applying them directly on instances of data elements rather than the schema. So when one drags a set of elements and drops them on another set, the specified operation is performed dynamically. Examples of such operations are Join, Semi-Join, Insert and Delete. The choice of these operations will be determined by the designer to set the requirements of the application. Here again, the designer states them declaratively and need not program them for each application.

2. Inducing a Query from a Sequence of Navigations

In this model, a query becomes a sequence of navigations and operations on data. In order not to repeat the sequence, the system should be able to automatically induce the query from the sequence of navigational operations and allow the user to name it and parameterize it for later use.

As mentioned earlier, the focus of this paper was to informally describe the Interaction Abstractions. Here we will briefly elaborate on the remaining three areas without going into much detail.

Area 3: Presentation Abstractions

Although various widgets can be used to hide large amounts of data, the limited screen can very easily and quickly become cluttered with widgets. To mitigate this problem, various presentation abstractions can be introduced.

a. An abstraction that allows the user to specify pop-ups to be triggered on certain conditions
b. Abstraction for automatically mutating widgets when the data set gets larger or smaller.

Area 4: Logic Abstractions

In addition to displaying values from data sources, the designer should have enough abstractions to:

a. Derive fields from instances.
b. Set aggregate data from a set of instances
c. Use decision trees for expressing simple logic

It should be noted that for the class of applications that we are considering, we are assuming relatively simple logic expressions. Anything beyond that should be done by some extensions to the underlying system.

Area 5: Data Source Abstractions

When the data source is structured like ODBC data source, the designer visually connects the attribute data source to the desired widget. On the other hand, if the data source is less structured, like spreadsheet data or pure text found in web pages, we need abstractions to perform the mapping through specifying various tags to induce the structure. We are currently working with Concordia University, Montreal, Canada, (10) to define an engine and algorithms to map spreadsheet data into relational schema. This is needed to define user gesture abstractions to specify the visual mapping of the data from a spreadsheet to a relational database.

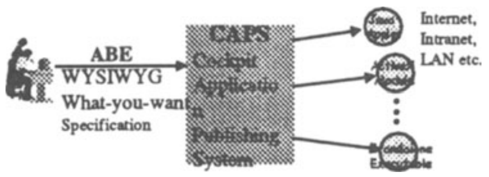

Fig. 4: CAPS System

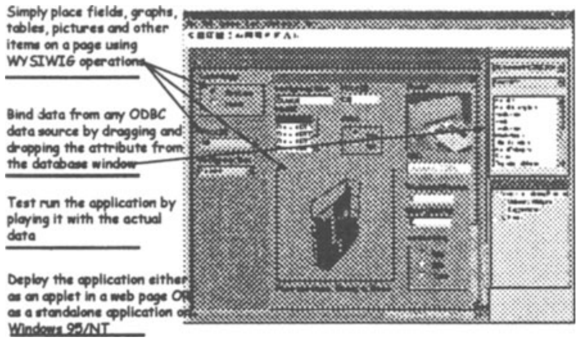
Simply place fields, graphs, tables, pictures and other items on a page using WYSIWIG operations
Bind data from any ODBC data source by dragging and dropping the attribute from the database window
Test run the application by playing it with the actual data
Deploy the application either as an applet in a web page OR as a standalone application on Windows 95/NT

Fig. 5: CAPS Development Environment

At Hewlett-Packard Laboratories for the past five years, we have been working to define these abstractions (11) and build a system called CAPS, (12), to process the declarative language called Application-By-Example, Fig. 4. The system is currently demonstratable and Fig. 5 demonstrates an example of an interactive screen that was constructed without a single line of code.

In ABE, a GUI program is constructed as follows:

1. Place the widgets on the screen: i.e., paint the screen
2. Hierarchically partition the set of widgets on the screen, i.e., LEVEL specifications
3. Declare the category of each level, i.e., SYNCHRONIZED, ANCHORED, REDUCED
4. Bind the widgets to the attributes from the database.

.This results in a rendering program that is devoid of events programming and other such procedural coding. Even though these steps may seem straightforward and simple, it is still a daunting task for a user to conceptualize his/her GUI applications in terms of the above abstractions such as level categories, widget classifications, etc. We address this problem next.

WYSIWYG PROGRAMMING ENVIRONMENT

There are two major problems with programming:
1. Where to begin? This is the age-old writers' block problem faced by any author. The designer has the same problem in deciding where to begin in constructing the GUI program.
2. Indirect programming: Programming has always been indirect in the sense that the execution of the program is the ultimate goal and a program is an indirect statement of that ultimate goal.
Consider text with HTML tags that represents a web page. Obviously editing/constructing this HTML file is an indirect specification of the ultimate web page. In contrast, a WYSIWYG web page editor is a direct manipulation of that page and a vast improvement over editing the HTML file. In our current implementation, we have addressed these two problems to enable web publishers to construct rendering applications easily.

We address the writers' block problem by letting the system suggest a GUI program that the user can edit through trial and error until the optimal goal is achieved. Initially, the user must only point at the attributes of the data source. The system analyzes the cardinality of each data source and suggests a suitable widget. For example, for a GENDER attribute, it will suggest a radio button widget. Similarly, if the cardinality is small, it may suggest to place the entire widget in a higher level than the rest of the data assuming the user may want to make a selection on these attributes, i.e., give me only the male or the female or both.
The latter is achieved by performing the modification to the running application in a WYSIWYG fashion.

WYSIWYG modification of a running application is the process of changing the rendering application to another rendering application. This is very similar to the spreadsheet programming wherein the formulae are edited directly into a spreadsheet that is a running application and the effect of the modifications are immediately seen. In a GUI application, the repertoire of WYSIWYG modification includes changes to the interactions with the widgets. If each such modification results in a new application program that also works and can be "test run", then the designer can make incremental changes until the final program matches exactly what they have in their mind. The well-known advantages of WYSIWYG editing are the direct specification of the changes to the application, immediate feedback of those changes and the ability for trial and error. All of these are very useful advantages that enable the web publisher to construct the rendering application that behaves exactly in the manner one wishes. Further, it also facilitates making incremental modification to the application to cope with future needs.

The WYSIWYG modus operandi is possible because every GUI program can be mutated by a sequence of modifications to any other program. This reachability property is mainly due to the fact that the declarative specification of level categories and widget categories allow all combinations with appropriate semantics for interactions. Therefore, mutating by changing level/widget categories as well as adding new level/widgets ensure that any application can be constructed through a series of mutations.

REASONING ABOUT THE PROGRAM

Since the designer constructs the program with declarative abstractions, it is easier for the system to reason at any time about the correctness of that program and generate dynamic messages along the construction process preventing the construction of ambiguous or incomplete programs. For example, if a level contains a single field, the system will automatically display a "back" and "next" button so one can reach other values in this field. But the moment a combo-box is placed in the level, the system deletes the "back" and "next" buttons, since all the data values are now accessible through the combo-box selection.
Another example: consider two widgets, one is multi-selectable like a list and the other is single selectable like a combo-box. If they are placed in the same level, the system will not allow the user to select more than one item in the list. Otherwise, it becomes ambiguous as to what item to display in the combo-box.

REFERENCES:

1. *Visual Basic*, Microsoft Corporation, Redmond, Washington
2. Cox, P.T., Giles, F.R. & Pietrzykowski, T. (1989) PROGRAPH: A step towards liberating programming from textual conditioning. In: *IEEE Workshop on Visual Languages*. Rome, Italy, 4-6 October, pp. 150-156.
3. Borges, J.A. (1990) Multiparadigm visual programming languages. Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign.
4. Ingalls, D., Wallace, S., Chow, Y.Y., Ludolph, F. & Doyle, K. (1988) Fabrik: a visual programming environment. *Proceedings ACM OOPSLA* '88. September, pp. 176-190.

5. National Instruments Corporation (1987) LabVIEW: a demonstration. National Instruments Corp., 12109 Technology Blvd., Austin, Texas 78727-6204
6. Shneiderman, B., "The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations", *Proceedings IEEE Symposium of Visual Languages*, 1996
7. Ahlger, Christopher, Williamson, Christopher, and Shneiderman, Ben, "Dynamic Queries for Information Exploration: An Implementation and Evaluation", *Proceedings ACM CHI*, 1992 Human Factors in Computing Systems, 619-626
8. Roth, S.F., Lucas, P., Senn, J.A., Gomberg, C.C., Burks, M.B., Stroffolino, P.J., Kolohehchick J.A. & Dunmire, C., *Visage: A User Interface Environment for Exploring Information, Proceedings '96 IEEE:* Information Visualization
9. Livny, M., Ramakrishnan, R., Beyer, K., Chen, G., Donjerkovic, D., Lawande, S., Myllymaki J. & Wenger, K., *Proceedings ACM SIGMOD* '97, DEVise: Integrated Querying and Visual Exploration of Large Datasets (DEMO ABSTRACT)
10. Lakshmanan, V.S., Subramanian, S.N., Goyal, N., Krishnamurthy, R., On Querying Spreadsheets, ICDE, February 1998, Florida
11. (KZ95) Krishnamurthy, R., and Zloof, M. M., "RBE: Rendering-By-Example", Intl. Conference on Data Engineering, Taipei, 1995.
12. (GHKMS 95) Goyal, N., Hoch, C., Krishnamurthy, R., Meckler, B., and Suckow, M. "Is GUI Programming a Database Research Problem?", SIGMOD 96, Montreal, Canada.

Biography

Dr. Moshe M. Zloof

Dr. Moshe Zloof, Principal Architect at Hewlett-Packard Laboratories, is considered a pioneer researcher in the area of data base languages and user interfaces. Back in the 1970's, during a complete departure from the traditional approach, Dr. Zloof created QUERY-BY-EXAMPLE (QBE), the first visual programming language which not only set the stage for considerable research agenda but also has been incorporated in many successful products such as PARADOX, DBASE IV, ACCESS and many more.

At Hewlett-Packard, Dr. Zloof is currently involved in developing the IC-BY-EXAMPLE language - a new paradigm to enable non-programmer professionals to construct their own applications.

Dr. Zloof started at IBM T.J. Watson where he worked for 13 years and became senior manager of office automation and visual programming. There he worked on QBE and later on OFFICE-BY-EXAMPLE (OBE). Subsequently he formed a start-up company to develop technology for PC software which was later sold to Ashton-Tate Corporation. He also joined Ashton-Tate and worked as a Chief Architect in Advance Development.

Dr. Zloof has published numerous papers and articles, has chaired and served as invited and keynote speaker at many national and international conferences and universities. He has also received several awards including the most prestigious IBM Corporate Award. He has also served as an adjunct professor at the Courant Institute of NYU and Columbia University.

Dr. Zloof received his BS. from the Technion Institute of Haifa, Israel, and his MS. and Ph.D. from the University of California at Berkeley in 1969 and 1972 respectively.