Developing an Advanced User Interface for HP VEE

Simplicity and flexibility were the primary attributes that guided the user interface development. Test programs generated with HP VEE can have the same advanced user interface as HP VEE itself.

by William L. Hunt

HP VEE, Hewlett-Packard's visual engineering environment, was developed as a programming tool for nonprogrammers. In the past, computer users were required to move into the computer's domain. Our goal for HP VEE was to bring the computer into the user's domain. This meant developing a system that operates in the way that our target users expect.

To accomplish this goal, ease of use was of critical importance. However, because most target users of HP VEE are highly educated technical professionals, simple-minded approaches to user interface design were not appropriate. For this audience, the system must be powerful and flexible, but must not become an obstacle because of overprotection.

With these constraints in mind, we decided that the primary attributes of HP VEE should be simplicity and flexibility. Learnability was also considered to be important, but we felt that no one would bother to learn the system unless it were a truly useful and powerful tool. Therefore, we felt that we could compromise some learnability in situations where a great deal of the power of the system would be lost if learnability were our primary goal. Our overall approach, therefore, was to design a system that is as natural to learn and use as possible and powerful enough that our customers would be excited about learning how to use it.

Development Guidelines

In general, simplicity is very important in a user interface because it frees the user from having to worry about unnecessary details or rules. The underlying goal of a good user interface is to increase the communication bandwidth between the computer and the user. However, this does not mean that there should be a myriad of displays and indicators. In fact, quite the opposite is true. The more things there are for the user to comprehend, the greater the chance that something will be missed. The goal, therefore, should be to reduce the amount of data that the user must be aware of and present the essential data in the simplest and most compact way possible. Similarly, any piece of data presented to the user should always be presented in a consistent way because this is known to increase comprehension dramatically.

An example of a simple way to present information to the user is the 3D look used in the OSF/Motif graphical user interface and more recently in other systems such as Microsoft® Windows. When used properly, the 3D borders can be used to communicate information about the state of individual fields without consuming any additional display space. Fig. 1 shows how HP VEE uses the 3D look to identify how fields will respond to user input. Fields that are editable are displayed as recessed or concave. Buttons and other fields that respond to mouse clicks are shown as convex. Fields that are only used as displays and do not respond to input are shown as flat. These states are very simple to comprehend because the three states are unique in the way that they look and operate. Without realizing it, users will naturally learn how to identify which fields are editable, which fields can be activated, and which fields will not respond to input. This 3D display technique allows these states to be displayed without any additional display area.

Fundamentally, HP VEE was designed around the concept of direct manipulation. This means that wherever possible, a setting can be changed by operating directly on the display of that setting. This results in a significant simplification for the user since special operations or commands are not generally required to make changes to settings. For example, the scale of a strip chart is shown near the edges of the graph display (Fig. 2). If the user wants to change the graph scaling, the limit fields themselves can be edited. It is not necessary to make a menu choice to bring up a pop-up dialog box for editing the scale. In many other systems, making any change requires a menu pick. This effectively reduces a system to a command-line interface that happens to use a mouse and menus instead of the keyboard. Such a system is no easier to use than the command line interface systems of the past.

Flexibility is more important for an easy-to-use system than for more traditional systems because there is a perception that power and ease of use cannot be combined in the same system. In the past, powerful systems have generally been

Fig. 1. A view containing a noneditable field, two buttons, and some editable fields.

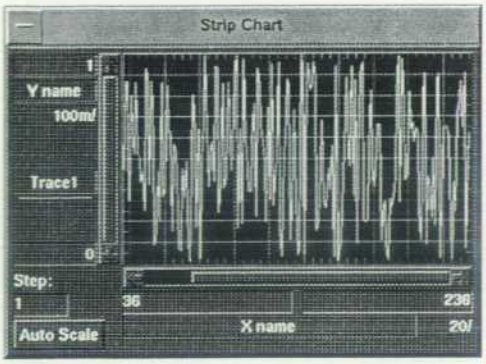

Fig. 2. Direct manipulation is useful for settings such as graph limits.

hard to use, and easy-to-use systems have generally not been very flexible or powerful. To overcome this perception, therefore, an easy-to-use system must be very powerful so that potential customers' fears can be overcome. When designing HP VEE, we were very careful to avoid limiting flexibility wherever possible. It often seemed as if we were faced with a choice between ease of use and flexibility. However, with careful consideration of the alternatives, we usually found that the more flexible approach could be implemented with an easy-to-use interface.

Flexible and powerful systems are naturally very complex because there are so many features and capabilities to remember. In these systems, it is extremely important that each area of the system operate in a way that is consistent with the rest of the system because even the most advanced users are rarely familiar with all aspects of the system. Users must be able to rely on their experience with other parts of the system to help guide them through the unfamiliar areas. For this reason, consistency was an important guideline throughout the development of HP VEE.

High performance for interactive operations is critical because users will become frustrated using a product that is too slow. Very few users will be happy if they must wait an inordinate amount of time before a particular operation is complete. The allowable time for the system to complete a task depends on the nature of the task and what the user is likely to be doing at the time. For example, a key press should be echoed back to the user within about 100 ms or so. If it takes longer, the user may press the key again thinking that the system didn't get the first one. A pop-up dialog box or menu should appear within about 500 ms. Other tasks such as loading a file can take many seconds before the user will become annoyed because of sluggish performance. We created a list of about ten different interactive operations for which we felt that performance was an important goal. On all supported platforms, many of the operations in this list such as the pop-up menus and dialog boxes are completed within the required time. Unfortunately, there are still a few operations that are completed within the specified time limits only on the very fast HP 9000 Series 700 workstations. On the other hand, we have received very few complaints about interactive performance, so our time limits may have been overly stringent.

In some situations, such as saving a file to the disk, performance simply cannot be guaranteed. In these cases, continuous feedback indicating progress being made is important. Otherwise, it isn't easy to tell whether something is happening or not. In HP VEE, all user-invoked operations that could take more than about 200 ms will result in a change to the mouse cursor. Some of these cursors represent specific activities such as reading from or writing to the disk. For other situations, a general hourglass cursor is used. Any action that is expected to take longer than one or two seconds is also accompanied by a pop-up message box that indicates that the operation is in progress.

Reducing the total number of mouse clicks, menu choices, and various other adjustments required of the user was a major challenge. Each adjustment required of the user, no matter how easy, will reduce the user's overall effectiveness. For this reason, HP VEE is designed to do as much as possible with default settings while allowing adjustments if more control is desired. Other systems often require that the user fill out a form each time a new object is selected from the menu. In most cases, HP VEE will insert default values for all settings and then allow the user to change them later if it becomes necessary.

System messages for errors and other reasons are an especially important source of difficulty or frustration for users. Most software developers seem to take the attitude of a hostile enemy when presenting the user with an error message. However, errors are seldom true user mistakes, but instead are usually triggered by failings in the system either because it misled the user or because it did not adequately protect the user from making the mistake in the first place. In many cases in HP VEE, we were able to avoid generating errors simply by restricting available choices to those that would not result in an error. For example, if a certain combination of selections will cause an error, we show them as mutually exclusive choices. In cases where such restrictions could not be used to avoid the potential for an error, we worked to simplify the interface so that users would be less likely to make mistakes in the first place. In cases where errors were unavoidable, we kept the attitude that error messages should help the user complete a task. We tried to remember that the user generally has a valid reason for performing the operation that resulted in an error.

Two kinds of messages that are common in many systems are not present in HP VEE. The first is the message "Please wait." It is irritating to users because they don't want to wait and they are being instructed to do so. The message is also unnecessary since more descriptive messages can be used instead. Messages such as "Reading from file program2" are much more informative and are not nearly so annoying. The other common message is a confirmation box that asks "Are you sure?" This is also very annoying because the user seldom initiates any operation without being pretty sure about wanting to perform that operation. There are really two reasons for asking "Are you sure?" One is to caution the user about data loss and the other is to protect against accidental requests.

In HP VEE, we solve the first situation by asking a question such as "Save changes before clearing workspace?" This has the same result as "Are you sure?", but does not sound as if the user's choice (or sanity) is being questioned.

In the second situation, accidental requests are better solved by making the input mechanisms easier to operate without error or by making corrections easy to perform. For example,

instead of asking "Are you sure?" to find out if the user really wants to delete an object, HP VEE puts the deleted object into the cut buffer so that if the user decides that a mistake was made, the paste operation can be used to restore the deleted object.

Attention to detail is very important to a user. Most systems available today lack the small details that make a system more convenient and easier to use. In HP VEE, we have attempted to pay attention to as many of these small details as possible. For example, when connecting a line to a box, an outline is displayed around the pin that would be connected if the line were released at that point. Another example of a very small detail is that HP VEE allows objects to be resized as they are being placed on the workspace for the first time. These seemingly minor details help reduce the amount of frustration that users often feel.

Program Visualization Features

In a traditional programming environment, the programmer must spend a large fraction of the development time thinking about details of the programming process including the language syntax, debuggers, and so on. Since HP VEE allows the user to think almost exclusively in terms of the problem domain, most of the development time and effort is spent on solving the problem instead of the programming details. This is the primary source of the productivity gains that many users of HP VEE have experienced. However, even though HP VEE allows programs to be expressed directly in terms of the problem, not all user-written programs will run correctly the first time. Although the so-called accidental complexities[1] of program development such as language syntax and semantics have been reduced or even eliminated, the fundamental complexities of the problem itself still remain. Therefore, once an HP VEE program is developed, it is likely that some aspect of it will not quite work as expected. For this reason, we developed several tools that can be used to visualize the execution of a program to help identify the source of any problems.

Show Execution Flow animates the execution of the program by outlining each object as it begins to execute and then erasing that outline when execution is complete. Besides helping to visualize how the program executes, this is useful when trying to understand performance issues, since an object in the program that consumes a lot of time will be highlighted for more time than other objects. HP VEE also has a timer object, which allows a more objective way to measure performance.

Show Data Flow animates the movement of data as it travels between objects in the program by displaying an icon moving rapidly along each line as data flows across it. This helps the user visualize the relationships between the data and the execution of the objects of a dataflow program. Both Show Execution Flow and Show Data Flow slow the execution of HP VEE programs so much that they are designed to be turned on and off separately.

All data in HP VEE has additional information such as size and shape associated with it. This information is maintained so that one operation can work with a variety of different data types and shapes. For example, math functions such as sin( ) can operate on either an individual number or an array of numbers with any number of elements. This is possible because the size and number of dimensions are packaged with the data. Other information such as the name of the data and mappings (the implied domain of the data) can also be associated with the data. The line probe feature allows the user to examine the data and this associated information at any time.

The execution of a program can be halted when execution reaches a particular object simply by setting that object's breakpoint flag. Breakpoints can be set on any number of objects at any time. When execution reaches an object with its breakpoint flag set, the program will pause and an arrow pointing to that object will appear. At that point the step button can be used to single-step the program one object at a time or the line probe can be used to examine data.

If an error occurs during execution of the program and no error recovery mechanism has been attached, a message will be displayed and an outline will highlight the source of the error visually. This allows the user to locate the source of the error more quickly.

User Interface for HP VEE Programs

Since a user of HP VEE should be able to generate programs with the same advanced user interface as HP VEE itself, several important capabilities have been incorporated into HP VEE to make the task of building impressive-looking programs simple.

For example, data can be entered using a variety of data entry objects. The simplest of these is a text field that accepts a single line of textual data. Numeric fields of various types such as integer, real, complex, or polar complex accept the appropriate numeric data. In addition, these numeric fields can accept constant expressions such as "SQRT(45)" or system-defined constants such as "PI." When typed, these constant expressions are immediately evaluated and the result is converted to the expected type by the input field. Since all editable fields in HP VEE share the same editing code internally, any numeric field in the system that requires a numeric entry can also accept a constant expression.

There are other more advanced mechanisms for entering data or specifying selections to an HP VEE program. Integer or real numeric input can be generated within a predefined range by using the mouse to drag the handle of a slider object. Selections from a list of acceptable values can be made using an enumerated list box, which can be displayed as radio buttons, as a single button that cycles through the list of values, or as a button that accesses a pop-up list box of choices. An HP VEE program can be designed to pause until the user is ready to continue by using the Confirm button.

Data can be displayed in a variety of ways. In addition to textual displays, real or integer numbers can be displayed on a meter object, which can show visually where a number falls within a range. Graphical displays such as XY graphs and polar plots show two-dimensional plots of data and can be interactively scrolled or zoomed. Stripcharts graph a continuous scrolling history of the input data.

All of these input and output types would have limited value if they could only be displayed when the rest of the HP VEE program with all of its lines and boxes is also visible. For this reason, HP VEE is designed with a feature called user panels, which allows an advanced user interface to be attached to a user-written HP VEE program. The user panel is built using an approach similar to many of the available user interface builders. Elements to be placed on the user panel are selected from the HP VEE program and added to the panel. The user can then move and resize these elements as appropriate for the design of the panel. Other layout options such as whether a title bar appears can also be adjusted. Since the elements on the user panel are selected from the user's program, no external code is required and the finished program is easier to build than with most user interface builder tools.

Another important aspect of an advanced human interface is the ability to hide data until the user has asked to examine it. HP VEE is designed with a feature called Show On Execute which allows HP VEE programs to use pop-up windows to hide data until a user request is received. This works by associating a user panel with a user object (similar to a subroutine in traditional programming languages) and enabling the Show On Execute feature. When the user object begins executing, the associated user panel is automatically displayed. When execution of the object is complete, the user panel is erased. Messages such as "Writing test results to file" can be displayed using this mechanism simply by putting the appropriate message on the associated user panel. If it is desirable to pause the program until the user has finished examining the displayed panel, a confirm object can be used.

Programs developed in HP VEE are highly malleable; they can be changed and adjusted as much as desired. However, in many situations it is desirable to protect the program from being changed. The secure feature in HP VEE accomplishes this by displaying only the user panel and making it impossible to alter the program or even look at it after the program has been secured.

Fig. 3. Simplified class hierarchy of HP VEE.

Using all of these features allows users to generate complete application programs with professional appearances without having to work outside of the simple dataflow environment.

Internal Architecture

Fig. 3 shows a simplified class hierarchy for HP VEE showing some of the key classes in the system and how they relate to each other in the inheritance hierarchy. The Object class is at the root of this hierarchy and implements the fundamental protocol for all objects in the system. This includes creating, freeing, and copying objects. The key subclasses of Object include View, which maintains a rectangle on the display, Container, which holds a unit of data, and Device, which represents the inner workings of an element in an HP VEE block diagram.

The fundamental visible element in HP VEE is implemented with the class called View. It maintains a single rectangular region on the display and can be transparent or a composite of other views. The View3d class adds a solid background color and a 3D border to View.

Views are organized into a hierarchy tree based on the display stacking order. The root of this tree is called DispDriver. It is always mapped to overlay the system window allocated to HP VEE. It performs all low-level screen display operations such as drawing lines and filling regions. It also handles the window system interface functions such as repaint requests and dispatching of input events. Fig. 4 shows a composite of views in a view hierarchy with some of the views labeled with the name of their associated class. Fig. 5 shows the complete hierarchy tree of the views in Fig. 4.

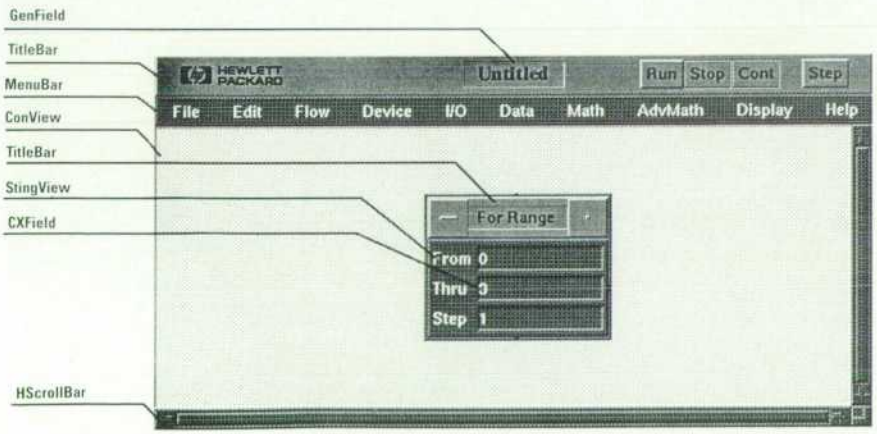

Fig. 4. A composite view with some of the component views labeled.

Fig. 5. Display hierarchy tree.

Subviews are views that are attached to another view called the superview in the display hierarchy tree. Subviews are clipped at the edges of their superview. In this way, each level of the view hierarchy tree limits the visual boundaries of all views below it. This view hierarchy indirectly describes the view stacking order, which ultimately controls which views appear to be on top and which ones are hidden.

Each view maintains a description of the region on which it is allowed to display itself. This clip region is calculated by taking its own bounds, subtracting any region that falls outside the bounds of any view in its superview hierarchy, and then subtracting any views that partially or completely cover it or any view in its superview hierarchy.

Repainting

When repainting an area that it is maintaining, a view may either use the clip region to limit the areas it actually changes on the display, or it may paint any area that it owns and then paint every view that appears closer to the user in the view stack. The full view stack repaint method has nothing to calculate or check before it begins painting itself completely and then painting anything that might be on top of it. If nothing is on top of it, then the complete stack repaint is very efficient because it is so simple. However, if there are many other views covering the view to be repainted, the full stack repaint will be very slow because of all of the unnecessary repainting that must be done. The clip region repaint method is much more efficient in this situation since only those areas that are directly visible to the user will be repainted. This means that no unnecessary repainting must be done.

Unfortunately, the clip region is at best an approximation since views are allowed to display images of arbitrary complexity (such as text) and be transparent in other areas. This gives the user the illusion that views can have any shape without incurring the performance penalties associated with nonrectangular views. It would be very costly to calculate these regions accurately, so only the approximation based on the rectangular view bounds is actually calculated. This means that repaints using the clip region method will not correctly update regions behind transparent areas of other views. Therefore, the clip region repaint method is used in only a few special cases.

Input events such as mouse clicks are dispatched to individual views in the system using a two-phase search mechanism. In the first phase, working from back to front, each view in the view stack where the event occurred asks the views in front of it to process the event. When there are no more views in front of the current view, the second phase begins with an attempt to consume the event. Working from front to back, each view in the view stack (as determined during the first phase) is given an opportunity to consume or ignore the event. If the view takes no special action, the event is passed to the next view down in the view stack. If the the view intends to consume the event, it does so by performing an action associated with the event such as indicating that a button has been pressed and then marking the event as consumed. This process continues until the event is consumed, or until the DispDriver class is given the event (this class consumes all events).

Other Classes

The visible part of each object in an HP VEE program is maintained by the DevCarrier class. DevCarrier's job is to maintain the visual appearance of all objects in the system by showing the terminal pins, maintaining the various highlights and outlines required by HP VEE, and allowing various editing operations on the user's program such as connecting lines and adjusting the sizes or positions of objects. DevCarrier does not display any object-specific information. That information is displayed by object-specific view classes, which are attached to DevCarrier as subviews.

User objects are specialized objects that are built using a subclass of DevCarrier called SubProg. SubProg maintains a visual subprogram which acts just like a normal object when viewed from the outside, but contains an internal dataflow network of its own that is just like the main program. All of the dataflow networks in HP VEE are maintained by a class called ConView (context view). It is the gray background area behind the lines and boxes in a dataflow network.

The top-level workspace environment class—IPEditor (iconic program editor)—is just a special case of SubProg and is therefore built as a subclass of SubProg. It is attached as the only subview of DispDriver and maintains the top-level editing environment. It is the same as SubProg, except that it must maintain the menu bar, run/stop buttons, and other features specific to the top level.

The context view class (ConView) maintains a list of all objects in the context and the lines connecting them. When the context view is asked to repaint itself, it first paints its background color (gray, by default), and then paints all lines in the line list. Then each HP VEE object in the context is painted according to the stacking order. If an HP VEE object falls partially or completely outside the context view's bounds, then according to the clipping rules, that view will be only partially painted or not painted at all.

The clipping and repaint algorithms allow an HP VEE program to be visually much larger than the screen space allotted to it. By adding navigation controls such as the background scroll capability, a very large dataflow network can be supported even on a comparatively small screen.

Model-View Architecture

HP VEE is organized around a model-view architecture. This is similar to the model-view-controller architecture used in other object-oriented systems except that we chose to merge the functionality of the controller into the view. The fundamental assumption in the model-view architecture is that the internal data and program elements (the models) can operate without any knowledge of or dependence on their visual representations (the views). By separating the system at this natural boundary, both the views and the models can be written more simply without any unnecessary dependencies. One feature of this architecture is that one model can be attached to any of several different views without any special support in the model. For example, a model that contains a real number can be attached to a text field or to a meter. Since the properties of the number do not change based on how it is displayed, no changes are required of the class that holds the number. However, since there are few similarities between a meter view and a text view, they need not be built with one view class.

User panels are one area that really benefit from the split between models and views. When the user selects an HP VEE object such as a slider and asks that it be added to the user panel, several things happen internally to make that happen. First, if a user panel has not been created for this program or user object, one is created. The user panel class is similar in concept to the context view class, but without support for interconnections required for dataflow networks. Next, an instance of the PanelCarrier class is created to hold a copy of the object-specific part of the slider view. This copy is created from the original and attached to the new panel carrier and to the original slider model (which is not copied). The panel carrier is then attached to the user panel view.

One of the most significant architectural impacts of the implementation of user panels is the fact that there can be many independent views attached to the same underlying model at the same time. Because of this architecture, it is easy for panels from user objects to be added as a unit to higher-level panels. This allows the creation of complex panels consisting of grouped controls and displays.

The DispDriver class is designed to be the only place where calls to the underlying window system (such as the X Window System) occur. This allows the display driver to be replaced if appropriate when porting to a new platform. During development, for example, we used a driver written to talk directly to the display card of an HP 9000 Series 300 computer because it ran so much faster than the window systems. Now that very high-performance workstations are available, this is no longer necessary.

Printing is handled simply by replacing DispDriver with the printer driver class, which knows how to perform graphics operations on a printer. The information intended for the printer is just "displayed" on the printer and no special printer support must be developed aside from the printer driver itself. This also allows the print output to match the screen display very nicely.

Acknowledgments

Building an advanced user interface is really not difficult, but it takes a great deal of thought and perseverance. It also requires support from management. We were lucky on the HP VEE team because we had managers who understood the value of a good user interface. They encouraged the team to produce the best product that we were capable of even if the schedule would be put at risk. Of course, the team members themselves were very highly motivated to produce an exciting product. John Bidwell, the HP VEE project manager, provided the leadership and management support required for our success. He was able to resist the temptation to ship the product before it was ready, and kept all of the various team members focused on the goal of a truly easy-to-use product. Sue Wolber, Randy Bailey, and Ken Colasuanno each contributed to the overall usability of the system in each of their respective areas. Jon Pennington performed usability testing and provided most of the usability feedback during development.

Reference

1. P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer*, September 1987, pp. 43-57.