Session 2

Software Engineering Activities at SEI

Chair: **Clyde Chittister**, *Program Director of Software Systems, Software Engineering Institute, Carnegie Mellon University*

Serpent:
A User Interface Management System

Reed Little, Software Engineering Institute, Carnegie Mellon University
Len Bass, Software Engineering Institute, Carnegie Mellon University
Brian Clapper, Naval Air Development Center
Erik Hardy, Software Engineering Institute, Carnegie Mellon University
Rick Kazman, Software Engineering Institute, Carnegie Mellon University
Robert Seacord, Software Engineering Institute, Carnegie Mellon University

Abstract

Prototyping has been shown to ease system specification and implementation, especially in the area of user interfaces. Other prototyping approaches do not allow for the evolution of the prototype into a production system or support maintenance after a system is fielded. This paper presents a set of goals for a modern user interface environment and Serpent, a prototype implementation that achieves these goals.

Introduction

The advent of the modern graphics-oriented workstation is placing increasing emphasis on quality of the user interface. End users are increasingly more demanding that software should be both functional and easy to use. In response, both software and hardware vendors must pay more attention to the user interfaces that accompany their products. However, it is very time consuming and expensive to construct a user interface: in some systems, the user interface development and maintenance cost exceeds 50% of the total software cost [1]. And if history is any indication, this cost is going to get more expensive in the future. The trend is to make these systems more "user friendly", which implies that the user interface needs to be more complex and robust, and thus more costly.

The current state-of-the-practice in the specification, design, implementation, and maintenance of interactive computer systems usually does not give the user interface of the system sufficient consideration. In general, software engineering techniques currently used for the development of systems are usually an ad hoc combination of "tricks" and "tools", with little regard for formalism and standardization. Further, the process of user interface development is labor-intensive. Current user interface development tools and methods inadequately address this problem. In particular, while more and more vendors are providing user interface toolkits and graphics packages, these packages typically require extensive and specific knowledge of a particular toolkit or user interface library. These packages also require the use of conventional, procedural languages such as C and Ada. These languages are not particularly well-suited to user interface specification and implementation, so the user is forced into worrying about low-level syntactic issues.

The Case for Evolutionary Development

One major problem with the software engineering of a user interface is that it is difficult to design a user interface and know a priori (before implementation) if it is "good". In fact, there are generally multiple, and often conflicting, definitions of "good". Some of the criteria used in the definition of "good" are:

1. does the operator "like" it?
2. does it support the mission goal? and

This work was sponsored by the Department of Defense.

3. is it fast enough?

The current methods used to build interactive systems can result in user interfaces that are non-intuitive for the operator to use and sometimes do not perform the necessary functions. Additionally, the user interface is often intertwined with the non-user interface parts of the system, making the task of modification and extension of the user interface during the sustaining engineering phase of the system extremely difficult.

In many respects, the user interface component is no different from the other components of a system. The user interface benefits from the accepted software engineering techniques, such as the determination of the specification of what is to be done before the design of how to do it, etc. However, user interfaces are especially difficult to build, and using a standard sequential method of construction (commonly known as the water-fall method) is not appropriate.

Practice has shown that it is better to use an iterative method, where there is specification, design, implementation, test, evaluation, and a return to specification again [2]. Frequently, there are several iterations of the specification to evaluation path. It is a fact of human nature that it is easier for people to determine what it is that they do not like about a user interface than it is for them to unambiguously specify what they want in a user interface.

Previous User Interface Approaches

Early prototyping efforts were marked by intense coding in traditional programming languages of both the user interface and the underlying application. This approach is cumbersome and error-prone, due to the low-level semantics of these languages. Using this process, changes to the user interface specification may force major changes in the application program. Even though the prototype may have only addressed some limited portion of the overall requirements, there is a natural tendency to use it as a basis for the deliverable product.

Later, specialized prototyping languages were developed, employing specific shorthand notations to generate corresponding function invocations [6]. These languages are usually fairly arcane, not unlike RPG and its successors, in that the user interface designer must be intimately familiar not only with the language, but also with the built-in functions. One of the big drawbacks to this approach is that after the prototype has been built, the user interface must be recoded (using the prototype as requirements) due to the performance and maintenance issues; there is no smooth transition from prototype to product.

With the advent of fourth generation languages and the increased use of computers for management information systems came the concept of rapid prototyping [4]. This approach is marked by the application of database concepts to software development: changing a value in the database causes a resultant change in the presentation. One major advantage over other approaches is that, for each function that can be invoked by the user, there is a corresponding program-callable routine. Once the user interface is specified, the appropriate calls can be made by the application program. However, if the user interface changes, the application program must be changed.

The explosion in workstation capabilities in the last few years has sparked many new ideas about how to use these capabilities for user interface development [9, 10, 8, 3, 5, 7], leading to a multitude of tools and environments, such as Prototyper, XVT, UIL, Granite, Autocode, and MIKE. However, each tool is marked by the use of a specific language and/or interactive tools tailored to the capabilities of a particular

platform and/or to the specific user interface toolkit supported. Application support in these packages usually takes the form of a fixed set of functions that can be invoked as necessary by the application, or a set of functions that are dynamically generated by the prototyping tool to implement the user interface. Again, if the user interface changes, the application must be changed to invoke the new functions.

Finally, user interface technology is evolving rapidly. Today's leading edge data presentation theory becomes tomorrow's commonplace toolkit, giving way to some previously unimagined technology. None of the above approaches adequately provides for the effective integration and use of new toolkits.

Goals of a Modern User Interface Environment

In 1987 the Software Engineering Institute started the User Interface Project to address perceived problems in user interface development and to assist the transition of user interface design and development technology into practice. Out of this effort arose a set of goals for the next generation of user interface environments:

1. In any computer system, there should be a true separation of concerns between the application and the user interface. This is simply the concept of modularity: the application should not try to perform the functions of a user interface, and vice versa. One should be able to develop the application independently of the user interface, in a language appropriate to the semantics of the application; similarly, user interface development should be independent of the application.

2. The user interface specification, design, and implementation should be simple and straightforward; prototyping should be fairly easy using the mechanisms provided by the environment. Non-programmers should be able to perform these activities with a minimum of training. The mechanisms used to perform these activities should not have to change, even though the user interface style or underlying user interface toolkit may change.

3. It must be possible to prototype the interface and functionality of a system without an application. The user interface support mechanisms should be sufficiently rich to support reasonably sophisticated prototypes. As the prototype matures, facilities should be provided to add an application, in pieces or all at once, thus providing evolutionary development.

4. Existing systems should be able to take advantage of new toolkits as they become available, without affecting the application portion of the system. The mechanisms for incorporating these new toolkits should be relatively simple.

5. Performance, when the environment is used strictly as a prototyping vehicle, should be reasonable, although special performance considerations may have to be made when used in production.

User Interface Management System (UIMS)

One tool which meets the above goals is the UIMS. A UIMS is generally composed of four parts:

1. a dialogue, which specifies how information is to be presented to the operator and how to respond to operator commands,

2. a dialogue manager, which is responsible for interpreting the dialogue during the execution of the system,

3. a realization component, which is responsible for the actual physical interface between the operator and the system, and

4. the application, which is responsible for all the non-user interface functionality of the system.

A UIMS can be thought of as software oriented "erector set" that is tailored for the development of user

interfaces. The UIMS provides an environment where it is very easy and fast to change the form and function of a user interface. This provides the ability to quickly prototype and change the user interface during the system specification, design, and implementation phases. A UIMS also enforces the separation of what is to be presented to the operator from the how it is presented. This provides a very convenient mechanism for the decoupling of the user interface from the rest of the system, which makes maintenance and the changes to the user interface easier.

Serpent

Starting with the above goals, the User Interface Project developed a user interface environment known as Serpent. Serpent is a UIMS, using the standard Seeheim model [11], that supports the development and execution of the user interface of a software system. Serpent supports incremental development of the user interface from the prototyping phase through production to maintenance. Serpent encourages the separation of concerns between the user interface and the functional portions of an application. Serpent is easily extended to support multiple toolkits.

Architecture

Figure 1 shows the overall architecture for Serpent. The architecture is intended to encourage the proper separation of functionality between the application and the user interface portions of a software system. The three different layers of the architecture provide differing levels of control over user input and system output. The presentation layer is responsible for layout and device issues. The dialogue layer specifies the presentation of application information and user interactions. The application layer provides the actual system functionality.

The *presentation layer* controls the end-user interactions and generates low-level feedback. This layer consists of various toolkits that have been incorporated into Serpent. A standard interface has been defined which simplifies adding new toolkits. Each toolkit defines a collection of interaction *objects* visible to the end user.

The *dialogue layer* specifies the user interface and provides the mapping between the presentation and application layers. The dialogue layer determines which information is currently available to the end user and specifies the form that the presentation will take, as previously defined by the dialogue specifier (the individual responsible for creating the user interface specification, or *dialogue*). The dialogue layer acts like a traffic manager for communication between application and toolkits. The presentation level manages the presentation; the dialogue layer tells the presentation what to do. For example, the presentation layer manages a button that the end user can select; the dialogue layer informs the presentation layer of the position and contents of the button and will act when the button is selected.

The *application layer* performs those functions that are specific to the application. Since the other two layers are designed to take care of all the user interface details, the application can be written to be presentation-independent; there should be no dependency in the application on a specific toolkit.

The data that is passed between different layers is known as *shared data*. Data passed between an application and the dialogue layer is referred to as *application shared data*, while data passed between a toolkit and the dialogue layer is called *toolkit shared data*. A *shared data definition* provides the format of the data.

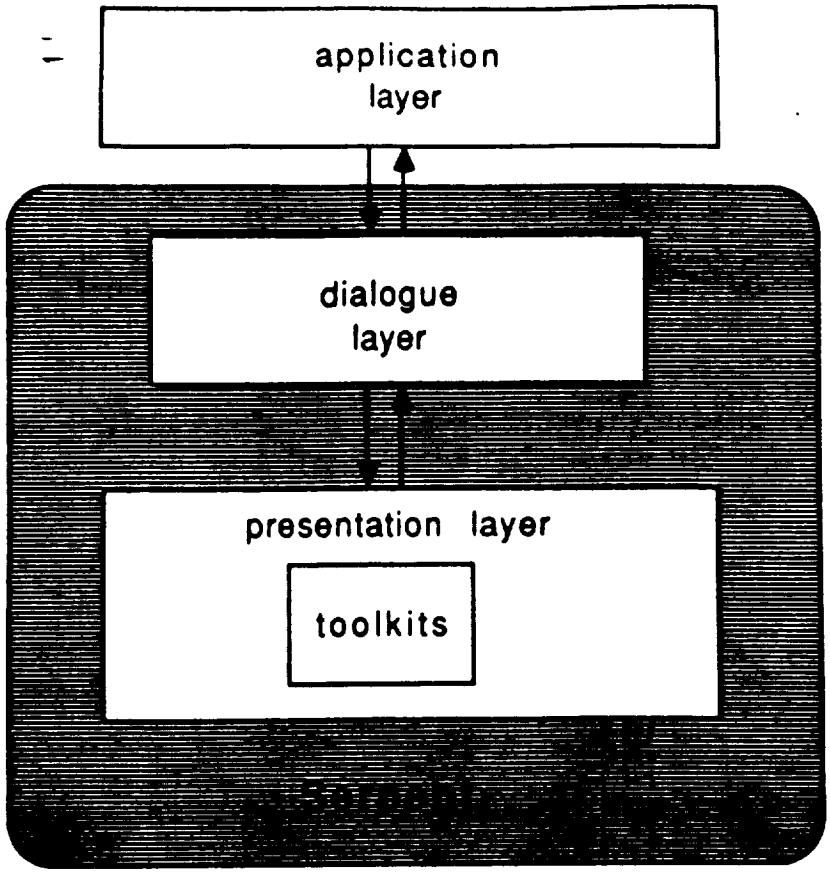

Figure 1: Serpent Architecture

Slang

In Serpent, user interface dialogues are specified in a special-purpose language called Slang. Slang provides a mechanism for defining the presentation of information to, as well as interactions with, the end user. A Slang program defines and enumerates a collection of interaction objects and allowable actions to be available to the end user. Slang provides variables for intermediate storage and manipulation, along with a full complement of primitive arithmetic operations.

The interaction objects available to the dialogue writer are defined by the toolkit. Each toolkit defines a set of primitive objects that may be used in a dialogue. Each object has a collection of *attributes* that define its presentation and a collection of *methods* that determine how the end user can interact with that object.

In Slang, dependencies between items are automatically enforced. That is, suppose variable V depends on the value of some object attribute A. If A changes (perhaps due to some end user action), the value of V is re-evaluated automatically. This important and powerful feature allows the dialogue writer to build complex, interdependent interaction objects simply by referencing data items; the dependencies are automatically determined and enforced by the the Serpent system.

Slang also allows a dialogue writer to group arbitrary objects into logical collections called *view controllers* that may be created or destroyed as a unit. Specifying a view controller in Slang defines a view controller *template*; each template has a *creation condition* that defines when an *instance* of the template should come into existence. The existence of a view controller instance and its child objects can be controlled by the values of Slang variables or by the creation, modification, or destruction of application data. When a view controller instance's creation condition is no longer valid, it and its associated objects are destroyed. Multiple instances of a view controller template may exist at any time. A view controller serves two main purposes:

1. It maps specific application data onto display objects with which the end user can interact.
2. It controls the existence of a series of related objects.

Application Program Interface (API)

From the application developer's perspective, Serpent behaves like a database management system. Shared data is a "common" database manipulated by the application, the presentation layer (usually in response to end-user actions), or the dialogue layer (in response to actions within the dialogue).

The application can add, modify, or delete shared data. Information provided to Serpent by the application is available for presentation to the end user. The application has no direct interface to the presentation layer and therefore cannot affect how data is presented to the user. When end user actions cause the dialogue to change the application shared database, the application is automatically informed. In this sense, the application views Serpent as an *active database manager.*

Saddle

The type and structure of data that is maintained in the shared database is specified in a *shared data definition* file, defined in a language called Saddle. This data definition corresponds to the database concept of *schema*. A shared data definition file is created once for each application and once for each toolkit that is integrated into Serpent.

The shared data definition file is processed to produce a language-specific description of shared data. Processors currently exist for Ada and C. If the application is written in C, the processor will generate structure definitions that can be included into the application program. If the application is written in Ada, the processor will generate package specifications.

Input/Output Toolkit Integration

Given that Serpent manipulates objects, the toolkits that are integrated most easily are those that are object-oriented. The successful integration of object-oriented graphics systems and their associated toolkits has been a major proof of Serpent's ability to separate presentation concerns from application concerns.

The process of integrating a toolkit into Serpent is conceptually simple. It can be logically divided into three parts:

1. the objects with which the end user will interact must be determined, along with their behavior;
2. these objects must be defined to Serpent through the use of Saddle; and
3. "glue" code must be written to allow the toolkit to communicate with the dialogue manager, through Serpent's shared database facility.

If a toolkit already has an object orientation, then the first and third integration steps are usually

straightforward. If it does not, then a set of objects and their attributes which conform to the Serpent model must be built on top of the toolkit.

Toolkit integration presents other practical difficulties. The integrator has to decide how much of the underlying toolkit to expose to a dialogue writer, whether to change any of the default behavior of the system, and whether to make the system more robust by, for instance, performing error checking that the toolkit does not handle.

The User Interface Development Process Using Serpent

Slang was designed explicitly for user interface specification. A Slang dialogue writer is not burdened with the technical and procedural details necessary to manipulate specific interaction objects; those details are hidden in the presentation layer. The dialogue writer merely specifies the objects that make up the user interface and indicates how they relate to one another and to the end user; the Serpent runtime system manages the interaction objects. The dialogue specifier needs to be familiar with the characteristics of various objects, such as knowing that an Athena widget set label widget appears as a rectangle on the screen; however, the specifier does not need to know how to tell the Athena toolkit library how to display such a widget.

Slang dialogues can be executed without of an application, allowing the building, testing, and refinement a prototype before designing and implementing the rest of a system. Often, however, a prototype requires the existence of some application functionality, if only to initialize display values. Slang's rich set of primitive operations allow the user interface designer to "mock up" application operations in the prototype dialogue. Once the prototype has been refined, the simulated application behavior is removed from the dialogue and the real application is added.

A Simple Example

Perhaps the best way to illustrate the simplicity of prototyping with Slang is by example. Figure 2 shows the screen display for a counter demonstration, using the X Toolkit Athena widget set. The box labeled "PRESS" is a command widget that can be selected by the user via a mouse. The box above the command widget is a label widget containing the current value of the counter. When the user selects the button labeled "PRESS", the value in the label widget is incremented by 1.

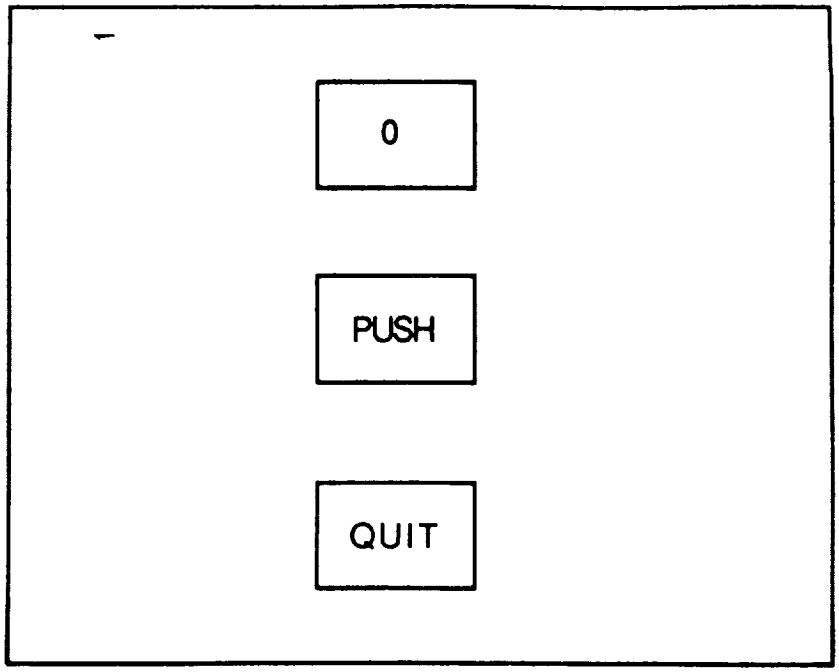

Figure 2: A Simple Example

In Slang this example is implemented as follows:

```
VARIABLES:
   counter: 0;

OBJECTS:
   /*
      width, height, vert_distance, and horiz_distance
      are all specified in pixels
   */
   background: form_widget
   {ATTRIBUTES:
         width:  640;
         height: 645;
   }

   display:  label_widget
   {ATTRIBUTES:
         parent:          background;
         width:           60;
         height:          40;
         vert_distance:   150;  /* from upper left of parent */
         horiz_distance:  310;  /* from upper left of parent */
         label_text:      counter;
   }

   push_button:  command_widget
   {ATTRIBUTES:
         parent:          background;
         width:           60;
         height:          40;
         vert_distance:   250;  /* from upper left of parent */
         horiz_distance:  310;  /* from upper left of parent */
         label_text:      "PRESS";

      METHODS:
         notify:
            {counter := counter + 1;
            }
    }
```

The **background** object provides a form on which to locate the other objects. The **display** object defines the label widget containing the current value of the counter; note that the **label_text** field, which controls what is actually displayed in the form widget, is dependent on the value of the global variable **counter**. When the value of the variable changes, all items that depend on it are re-evaluated. Put more simply, if **counter** changes, the text displayed in the **display** object will change automatically.

The **push_button** object defines the command widget that the end user will select in order to increment the value displayed on the screen. When the user selects the button, the presentation layer captures the event and communicates it to the dialogue via a *notify* method, causing the associated code snippet to be executed. In this case the **counter** variable is incremented, which in turn causes the label in the **display** object to be changed.

Dependencies and type conversions are managed automatically by the Serpent runtime system, allowing the dialogue writer to focus on user interface issues, rather than syntactic details. For example,

the **counter** variable is an integer; the **label_text** attribute of the **display** object is a string. Slang converts the **counter** value to a string before assigning it to the the **label_text** attribute; the dialogue writer merely needs to specify the dependence between the variable and the attribute. Further, the attributes for every interaction object take reasonable defaults, so the dialogue writer does not need to specify a value for every possible characteristic of an object.

In short, Slang is designed to minimize the amount of information the dialogue writer needs to specify in order to manipulate interaction objects.

Status

The initial implementation of Serpent was done under ULTRIX 2.2 on DEC microVAX II and III workstations. Serpent was also easily ported to run under SUNOS 3.5 or higher on SUN2 and SUN3 workstations and DECStation 3100 & 5000 platforms . We expect porting to similar UNIX platforms to be relatively straightforward.

Applications can be written in either C or Ada, and simple mechanisms exist to extend Serpent to support other high level languages. Serpent was implemented predominantly in C, with additional support software written as shell scripts.

Currently, two different interfaces to X Window System toolkits have been written for Serpent: one implements a subset of the Athena widget set and the other implements the Motif widget set. In addition, Lockheed's Softcopy Map Display System has been integrated.

An interactive What-You-See-Is-What-You-Get (WYSIWYG) graphical editor that hides most of the details of the user interface specification is available. The editor provides for fast feedback, so that the entire application system need not be executed, or even exist, to begin to "get a feel" for the interface.

Serpent is available from the Software Engineering Institute and MIT through anonymous ftp. It is also contained in the X11R4 contrib release from MIT.

Conclusions

As a result of our experiences in developing user interfaces with Serpent, we have concluded that Serpent offers the following advantages over other user interface development approaches:

1. The active database model for applications allows the true separation of application issues from user interface issues, thus ensuring modularity. Application writers are also free from the syntactic drudgery inherent in programming large, complex toolkits.
2. The constraint mechanisms implemented via automatic dependency updates ensure that all participants (application, dialogue manager, and toolkit) are synchronized in terms of the state of the system.
3. Serpent's language-independent interface definition and inter-process communication mechanisms help in achieving modularity. Application developers are not constrained to work in a single language.
4. Serpent's toolkit integration support reduces the integration process to a series of concise, well understood steps. Once a particular toolkit is integrated, its objects are available for use in any dialogue.
5. Due to Serpent's inherent separation of concerns, system developers can experiment with different user interface styles, and even different toolkits, without changing either the application code or the API. This also provides for the injection of new toolkits and user interface paradigms into an existing system, while minimizing the system portions which are

affected.

Serpent has achieved the goals of a modern user interface environment set forth earlier. The user interface specification mechanisms are simple and direct; changes in the user interface are made easily, without changing the application. The application program interface is simple and easy to use and enforces a true separation between the application and the user interface portions of the system. Prototyping is accomplished rapidly, with reasonable provision for application functionality simulation. Serpent's toolkit integration mechanisms allow a new toolkit to be incorporated into Serpent easily without affecting the application. Finally, Serpent is itself a prototype, implementing the goals listed above. Even so, performance is quite reasonable, and we are continually making improvements, although we would not yet recommend it for time-critical production environments.

References

[1] Boehm, Barry W.
A Spiral Model of Software Development and Enhancement.
Computer 21(5), May, 1988.

[2] Boehm, Barry W.
Improving Software Productivity.
Computer 20(9), September, 1987.

[3] Colborn, Kate.
OSF Determines User Interface; Choices Could Affect the Development of Applications Software.
EDN, December, 1988.

[4] Fisher, Gary E.
Application Software Prototyping and Fourth Generation Languages.
Technical Report, National Bureau of Standards, May, 1987.

[5] Foley, James, et al.
Defining Interfaces at a High Level of Abstraction.
IEEE Software, January, 1989.

[6] Hanner, Mark Allen.
Gambling on Window Systems.
UNIX Review, December, 1988.

[7] Kasik, David J., et al.
Reflections on Using a UIMS for Complex Applications.
IEEE Software, January, 1989.

[8] Kolodziej, Stan.
User Interface Management Systems.
Computerworld, July 8, 1987.

[9] Myers, Brad A.
Tools for Creating User Interfaces: An Introduction and Survey.
Technical Report CMU-CS-88-107, Carnegie Mellon University, 1988.

[10] Myers, Brad A.
The Garnet User Interface Development Environment: a Proposal.
Technical Report CMU-CS-88-153, Carnegie Mellon University, 1988.

[11] Pfaff, G. (Ed.).
User Interface Management Systems.
Springer-Verlag, Berlin, 1985.