Developing Applications with a UIMS

Daniel Klein
LoneWolf Systems
133 Lanford Drive
Pittsburgh PA 15235-1856

dvk@lonewolf.com

A User Interface Management System (UIMS) is a collection of programs and libraries that enables an application to be partitioned into functionally distinct component parts. This partitioning recognizes that any application is comprised of three types of components, each encompassing a separate domain of expertise. These are the application domain, the display technology domain, and the user interface domain. By separating the application into dialogue utilizing a collection of agents, development time is reduced and maintainability is enhanced. Because the dialogue views the agents as black boxes, the benefits of parallel development and independent maintenance and upgrade are realized. Display technology agents can be easily reused, enabling a corporate look-and-feel. Further, an application agent can be reused with different user interfaces, creating systems tailored to different classes of users. We describe the general features of the *Alpha* UIMS, and show how a complete application can be developed using such a User Interface Management System.

1. What Is a UIMS?

Except for the simplest, non-interactive programs, every program has a user interface. The user interface is that aspect of the program that communicates with the user. This communication can be by way of a keyboard, through a point-and-click mouse-based window system, with a keypad and touch screen, a voice recognition system, or a gesturing system. The future will doubtless reveal new technologies such as eye-tracking systems or even direct computer-to-nervous system implant interfaces. Science fiction is the harbinger of the science to come, and no interface technology is too far-fetched.

In general, programs today are written with the user interface sub-domain tightly coupled to the application code. Although this is standard practice, it presents numerous problems. For example, when a bank introduces a new banking machine, it makes no difference that they already have developed bank-in-the-box programs for the three previous versions of machine. The bank must usually contract out the development of not only the new user interface, but the new banking code that resides in the machine.

The truth, though, is that the application program is very much the same for all banking machines – all that is different is the interface. In fact, the application code is in all likelihood identical for a teller's workstation, too. The underlying CPU may be different, but this is a moot point considering portable languages and current compiler technology. Yet with all the common features of the old and new versions of the banking machine, an entire new system must be developed. Even if a small change to the existing interface is made, the changes to the overall system are often quite extensive. Why?

The problem is that the interface portion of the application is very tightly coupled with the non-interface portion. The main program calls routines that access the banking-related actions and also those that access interface-related actions. Moreover, these routine calls are typically intermixed. When any change to the interface is made, the programmer responsible might conceivably have to change the entire program! If the program has been well designed and written, the interface routines are modularized. In this way, the function "get account number" is separated from the rest of the program. More often than not, however, it is the "get keystroke" routine that is called throughout the body of code, so that the *function* of getting an account number is obscured by the *features* of getting individual keystrokes. Even if the application is properly modularized, the interface and display technology are still linked with the application domain – and it need not be!

An application that interacts with a user really consists of three things:

1) *The application domain code*. In the case of the banking system, this is the part of the program that understands deposits, withdrawals, transfers, and inquiries. All that it needs to know is that "customer X wants to perform transaction Y". If the system requires personal identification numbers (PINs), the application domain may also verify the validity of a customer-id/PIN combination. The application domain does not need to know how the account number is *entered* – it only needs to know what the account number *is*. It should not care whether the account number is typed on a teller keyboard or read off of a banking card. It also does not need to know about *how* the transaction request is acquired – only *that* a withdrawal of $100 has been requested.

2) *The technology-based code*. This is the part of the program that is concerned with *how* to display information on the screen, or how the buttons are read from the keypad and whether the numbers are displayed on the screen as entered, or how to read information from a magnetic card. It usually knows little or nothing about what the numbers or letters mean. It only knows that when a given button is pushed, some information must be passed to the dialogue (see below), and that some interactions with other technology-based objects may occur.

   Some higher level functions may be built into the technology layer. Depending on the level of functionality desired, the technology-based code may simply expose basic objects, such as buttons and scroll bars, from which the dialogue can construct complex aggregations. It may also expose more complex objects, such as a file selection menu, which searches through a file tree interactively with the user, only returning the final target file name to the dialogue.

3) *The user interface code* (also called the *dialogue*). This is the part of the program that is concerned with reading an account number (via the technology-based code) and passing it to the application domain code. This may entail reading the magnetic stripe on a banking card, or accepting a typed number from a teller, although the actual mechanics are done with the cooperation of the technology-based code. The user interface code may also be concerned with processing a high-level transaction, by informing the application domain code that "customer X wants to perform transaction Y". It does not need to know about how the transaction is processed or whether the account has sufficient cash in it. It only needs to know that a given sequence of technology-based actions signify that an application-based transaction is requested, and that an application-originated message of "give the customer $100" means that the technology-based money dispenser should be operated accordingly.

In a system implemented with "classical" techniques, these three portions are all integrated into a single program. The application, interface, and technology code are all intermixed (often egregiously). A change in any one portion invariably affects the others. It is possible, however, to separate the three portions. Rather than develop a monolithic application that performs all three functions, a better solution divides the program along functional lines.

A user interface management system (or UIMS) is a system that encourages and enforces this division [Pfaff85, Arch92], and provides a mechanism for allowing the three distinct aspects of the application to communicate with each other. The *Alpha* UIMS is one such user interface management system.† The *Alpha*

---

† Other UIMS's have been developed [Foley89, Kasik89, Kolodziej87, Myers88a, Myers88b], but remain predominantly prototypes or research tools.

UIMS builds upon the experiences and ideas of the Serpent UIMS [Bass90, Hardy91, Klein91], a research prototype developed at Carnegie-Mellon University. As a development system, the *Alpha* UIMS was designed to provide a clean, powerful, and fast implementation of a UIMS, while correcting the design deficiencies discovered in the seven years since the inception of the Serpent project. The *Alpha* UIMS also adds numerous functional enhancements to the original prototype system.

A UIMS is not a "builder", nor is it a "toolkit abstraction". Each of these were valuable evolutionary steps in application development. A builder allows the developer to draw pictures of interaction windows. The builder then generates the code necessary to rebuild the windows using the toolkit specified. The developer must still write the interaction code necessary to use these windows, and must also interface the application. A builder provides a mechanism for speedily developing the "look" of a user interface, but not the "feel" (the interaction between objects, and between the application and the windowing system).

A toolkit abstraction is a collection of library routines which interface between an application and the underlying toolkit performing the actual display actions. As such, it removes some of the dependence on a particular toolkit, but it also provides only the "look", and not the "feel" of an interface. Both builders and toolkit abstractions suffer from the shortcoming that the three parts of the application are still tightly coupled together. Changes to any part will often have affects in all the other parts.

A UIMS provides both the "look" and the "feel" (the presentation and interaction) of a user interface. It has all of the features of a builder, a toolkit abstraction, and more. By providing a clear separation between the three parts of an application, it also mitigates all of the problems of tightly coupling them together.

2. Implementation

An system written using the *Alpha* UIMS is implemented as a set of processes – zero or more application agents, zero or more technology agents, and exactly one dialogue manager. The dialogue manager executes the dialogue language program (written as a dependency-based collection of variables, objects with attributes, and collectors). The application agent(s) execute the user application, and the technology agent(s) execute the code needed to present interaction objects through their particular media. As far as the dialogue (and this document) is concerned, application agents and technology agents function in exactly the same way with respect to the dialogue manager. Their specific functions are different, but their mode of communication with the dialogue manager is identical. Unless a distinction needs to be made, we will refer to all technology agents and application agents simply as **agent**s.

By executing the dialogue program, the dialogue manager (or **dialogue**) acts as an intermediary between all agents. No two agents ever communicate directly. Rather, an action by one agent may cause indirect interaction with another agent by direct interaction with the dialogue. Actions of an agent that are exported (that is, that are communicated to the dialogue) may cause dialogue actions that cause other actions to be communicated to other agents. However, the only path between agents is through the dialogue, and then only through explicit dialogue language actions.

Figure 1 shows the execution model of running a dialogue. The dialogue program is executed by the dialogue manager. The dialogue can interface with any number of agents (which may be either application agents or technology agents). The interface between the dialogue and the agents is maintained by the communication library subroutines, and allows the dialogue and the agents to create or delete objects and change their attributes. The agents can also invoke dialogue messages, indicating special events.

3. Nomenclature

It may be convenient to think of the instances of objects as data which is "shared" between the agent and dialogue, although there is no "shared data" in the sense of shared memory. Briefly, the notion of data shared between components of the system is a cooperative one, in which both sides of an interface keep track of commonly referenced objects of a known format. Changes made by either party are communicated to the other party, and through continued cooperation, both parties separately maintain an up-to-date copy of the common data.

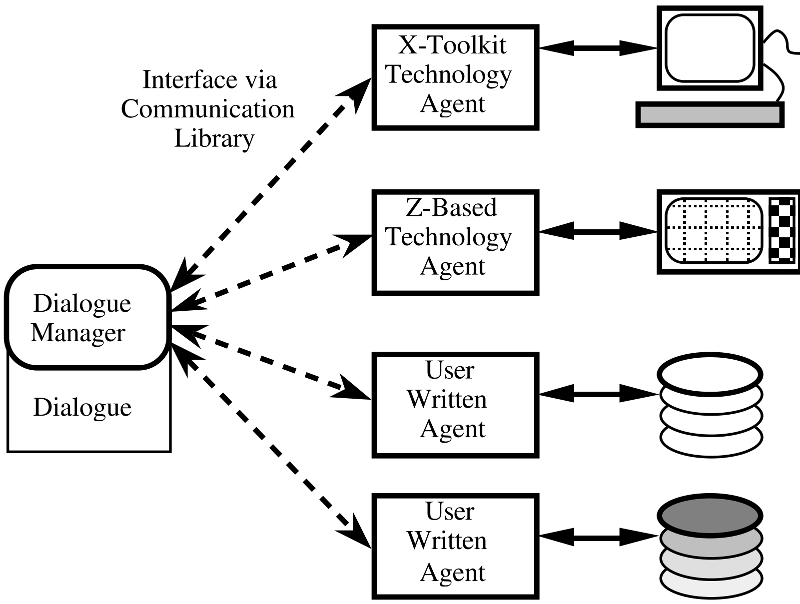

Figure 1 – UIMS Implementation

The types of objects that are “shared” between cooperating components are the objects defined in the interface definition file. These objects contain information that must be communicated between the agent and the dialogue manager. For example, in a X-based toolkit agent, an object might correspond to a push-button widget, or to a more abstract object type. In a banking agent, an object could correspond to a customer record or account. The *type* of the object is defined in an interface definition language, which creates language-specific notations to use with the communications package. The *instances* of the various types are created within the agent or dialogue programs, and the *contents* of an instance are communicated between the dialogue and agents, along with a notice that a new object instance has been created. Changes in the contents of an object instance can also be communicated, as can the deletion of an instance of a type. We use the term **object template** to describe the defining mechanism for an object. An **IDL file** (for **I**nterface **D**efinition **L**anguage file) describes a collection of interaction object templates. An IDL file defines the different types of objects that will be used by an agent. In general, a dialogue will use more than one IDL file, providing access to the templates for the objects associated with multiple agents.

The term **object** describes an instance of a template. There may of course be multiple objects instantiated from a single template. Objects are created statically or dynamically, depending on the agents actions.

An **attribute** is a component of an object. Objects are aggregate structures, with attributes containing a string, an integer, a floating point number, etc. Attributes can also be defined to be array of elements of a single type. A **message** is a special attribute of an object. It is used to signal the dialogue that a special condition exists within the application. A message differs from ordinary data in that invoking a message signals an *event*, whereas all other changes in an object’s attributes are *data* changes. When the agent invokes a message for an object, a special code snippet associated with the object can be executed.

A **collector** is a mechanism for grouping objects and variables according to a common existence condition. Collectors can exist solely based on a runtime condition, or they can be “bound” to another object. In this way, the dialogue can create a group of objects when another object is created. For example, a collection of buttons and labels can be created when a new customer object is created. A **stanza** is a special type of collector, bound to an instance of dialogue-specific data.

The **interface library** is a collection of C (or other language) routines which enables agents and the dialogue to communicate. Agent developers use the interface library explicitly to register or respond to the creation, destruction, or change to an object. The dialogue uses the interface transparently, so that changes to objects are reflected by the automatic execution of dependency-based and procedural code snippets.

The **dialogue language** is a programming language provided as part of the *Alpha* UIMS. It is one of the key elements of the *Alpha* UIMS, and provides features including typeless variables, dependency-based and procedural code elements, an automatic relational database, easy access to dynamically created interaction objects, collectors, and an interface to other programming languages, such as C. A **dialogue** is a program written in the dialogue language. When a dialogue executes, it uses user-written and system-supplied agents to create an entire system, communicating with the interface library. Unlike most other programming languages, a dialogue is a *reactive* program (or in other words, as an event-based system, where events are defined in terms of creation of, changes to, and destruction of interaction objects). A **snippet** is a portion of a dialogue program that is executed in response to a state change in the variables or object instances in the dialogue. Rather than initiating actions, a dialogue primarily responds to events in the application programs. Essentially, the dialogue is the communications manager between the agents, passing, changing, and regulating data that is shared between them. The dialogue also enables the user agents to interface with the various display technologies while separating the concerns of all the components of the system.

4. Pictorial Overview

The *Alpha* UIMS is divided into four conceptual components, shown in Figure 2, and explained in the list below it. Note that this figure is merely a simplified rendition of Figure 1.

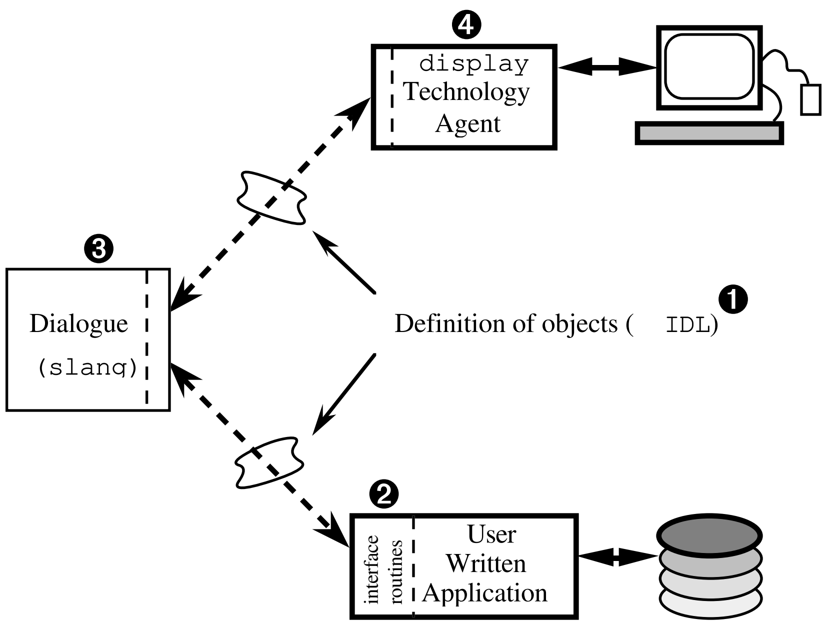

Figure 2 – Parts of a UIMS

1) The interface definition language (IDL), which is used to define object templates that are communicated between the dialogue and the various agents.

2) The interface library routines, which are used to effect communications between user-written agents and the dialogue. The interface routines are also transparently used by the dialogue.

3) The dialogue language, which is used to describe the layout and function of the dialogue, using the user-written applications and/or the *Alpha* standard technology agents.

4) Display technology agents. Although the *Alpha* user can create their own technology agents, at least one is supplied with the *Alpha* UIMS.

5. Building a System with the Alpha UIMS

To develop a system using the *Alpha* UIMS, the first thing that must be done is to decide how the system will be partitioned, and what objects will be supported by the various agents. The templates for the objects are defined using IDL.

Once the IDL file has been created, the dialogue and the agent(s) can then be written. The dialogue uses aggregate object notation to reference the IDL object types, while the agent uses the interface library routines. The dialogue can also reference existing agents, such as the `display` technology agent.

5.1. Partitioning of Control

One question that must be asked when developing a system using the *Alpha* UIMS is “where does the responsibility for control reside?” The reason for this question is that there is never a sharply drawn line between the responsibility of the dialogue and an agent – the line is flexible, depending on your design decisions. Here are two examples of this fuzzy line:

1) Consider the processing of a shipping address in an order-entry system. If the application agent has an “address” object, the dialogue can pass each part (street name, city, state, Zip code, etc.) to the agent as it is filled in by the user. The agent must then perform tests to determine if each part of the address is legal (and reject, for example, a four-digit Zip code) – the dialogue is then merely a data pass-through between the technology agent and the application agent.

   An alternative implementation is one in which the dialogue performs all of the validation tests itself, and only gives the application agent a fully filled-in address object. Either approach is a valid use of the *Alpha* UIMS. In fact, there are numerous intermediate implementations, where the dialogue and the application agent share the validation responsibilities.

2) Consider a technology agent, the program responsible for drawing on the screen. If the agent provides a mechanism for displaying a collection of “radio buttons,” this can be done in many ways. One way is to expose the underlying toolkit, so that Motif objects of type XmRowColumn and XmToggleButton are exposed, along with the attributes necessary to place the buttons on a pleasing layout (such as XmNradioBehavior, XmNradioAlwaysOne, XmNorientation, etc.). This means that the toolkit agent is simply a Motif interface, and the dialogue programmer must be intimately familiar with the details of Motif.

   An alternative implementation is one in which the agent knows all of the details of Motif (if indeed Motif is used at all!), and simply provides a canvas onto which exclusive buttons can be drawn. The toolkit agent worries about the specific toolkit calls. All the dialogue programmer does is say button #1 is to the right of button #2, etc. Each approach is “valid”, although the *Alpha* UIMS distribution takes the latter approach, and attempts to relieve the dialogue programmer of the odious task of becoming familiar with Motif.

6. The Interface Library

An agent programmer uses the interface library routines to communicate with the dialogue. Interface library functions fall into five categories:

1) Interface initialization and shutdown.

2) Object creation and destruction.

3) Object modification, including changing object attributes, and sending object messages.

4) Transport and status (bundling collections of actions into optimized packages for transport across the interface, targeting of calls, status messages, etc.).

5) Receipt and processing of changes from the dialogue.

The C language interface provides routines to call from a C-based agent – other language interfaces are also available. The file `time_of_day.c` in the appendix shows how the interface library is used.

7. The Dialogue Language (Slang)

The dialogue language (called `slang`) is used to describe a dialogue in the *Alpha* UIMS. A dialogue is defined as the interaction between the various agents which comprise a system. Typically, there will be a technology agent interacting with the user via a terminal, keyboard, mouse, etc., and an application agent doing the data manipulations in the application domain. The dialogue moves information between the agents, possibly modifying it *én route*, and possibly also making decisions as to when information will be moved. In this way, a dialogue interacts with interface objects defined by the various agents that it uses to provide a flexible interface to the user.

7.1. Mutable Types

The dialogue language does not require that variables be declared to be of a specific type. The type of a variable "mutates" according to the type and value of the operation which assigns it. Variable may be of type:

```
boolean  integer  real  string  id  bundle
```

Once a variable has been assigned a value and a type, the type remains unchanged until a new value and type are assigned to it. Some type conversions are automatic. For example, an integer expression may be used in a conditional statement – the integer value of the expression is converted to boolean before the conditional statement is evaluated. Special routines are also provided to copy the value of a variable and convert that copy to a specific type.

7.2. Dependency-Based and Procedural Code Snippets

A code snippet is a small piece of code that is executed given a set of conditions. The dialogue language supports two different types of code snippets – dependency-based and procedural.

Dependency-based snippets are executed when a specific *relationship condition* is updated. These relationships are defined to be changes in an independent portion of an expression. This is similar to the model provided by relational databases.

Procedural snippets are executed when a specific *trigger event* occurs. External triggers are typically object methods or creation actions. This is similar to the model used by most programming languages.

Dependency-based code is something which is unfamiliar to many programmers. However, the notion of data-dependency *is* something that is familiar. Any program which does something meaningful is predicated on the fact that things change. When changes occur, other things change in response (sometimes moderated by thresholds). Dependency-based programming expresses these relationships and cascades of data in a simple, straightforward manner.

7.2.1. C Example

Almost all high-level programming languages enable you to create data dependencies manually. Typically, you accomplish this by writing code that essentially states "this variable has changed, therefore I must also change the following". The nature of most languages, however, is such that however you state the dependency, you will make these statements *often*, and usually *redundantly*. Consider this simple example:

```
average = (left + right) / 2;
while (1) {
   /* main body of code goes here */
   if (changed (&left))
      average = (left + right) / 2;
   if (changed (&right))
      average = (left + right) / 2;
   }
```

In this example, we define a variable called `average` to be the average of `left` and `right`. Whenever either of the independent variables (`left` or `right`) change, we manually update `average`. In this way, data dependencies are expressed by asking the question "who depends on me?" The problem is that adding a dependency to a variable requires that you change the code associated with *all* of the independent variables in the dependency. If we want to change the definition of `average` to include a third variable called `center`, we have to update the code in four separate places to look like this:

```
average = (left + right + center) / 3;
while (1) {
   /* main body of code goes here */
   if (changed (&left))
      average = (left + right + center) / 3;
   if (changed (&right))
      average = (left + right + center) / 3;
   if (changed (&center))
      average = (left + right + center) / 3;
   }
```

This redundant updating of code can become very cumbersome and error-prone as the number of dependencies grows beyond that shown in this simple example.

7.2.2. Slang Example

A dependency-based programming language expresses data dependencies from the opposite viewpoint, by asking the question "on whom do I depend?" Dependency-base snippets define the data dependencies, which are reevaluated automatically whenever an independent variable changes. The first simple example we saw above would be written in Slang as:

```
average : (left + right) / 2;
```

This snippet *defines* `average` to be the average of `left` and `right`. Whenever either of the independent variables change, `average` is updated automatically. If we want to update the definition to include the third variable `center`, we simply change the code snippet to be:

```
average : (left + right + center) / 3;
```

Because each data dependency is stated in terms of what it depends on (instead of what depends on it), changing data dependencies is greatly simplified, and is substantially less error-prone that in conventional programming languages.

7.3. Independent Variables

Essentially, an independent variable is anything that could be considered a "right-hand-side" of an expression (or in C, an `rhs`). Because a dependency-based snippet may contain more than one statement (and each statement may contain complex expressions), a snippet may be dependent on more than one independent variable. For example, this snippet is dependent on both `alpha` and `beta`:

```
foo : { foo = alpha + beta }
```

If either `alpha` or `beta` changes, the snippet is automatically reevaluated. If a snippet only has a single statement, and that statement assigns the snippet "tag", it may be written using an abbreviated notation. The previous snippet can be abbreviated as:

```
foo : alpha + beta;
```

This abbreviated notation is the one most often used when writing a dialogue.

This next snippet is also dependent on both `alpha` and `beta`, although this snippet has two statements instead of one. If *either* `alpha` or `beta` changes, the *entire* snippet (both assignments) is reexecuted.

```
foo : {
   foo = alpha;
   baz = beta;
   }
```

8. An Example Dialogue

Figure 3 shows how the dialogue language can be used to construct a dialogue using a single agent. In this case, the agent is a technology based agent called `display`. This example is much simpler than an "real" application, and merely illustrates some of the features the dialogue language. A more complete example consisting of a dialogue, an application agent, and a display agent can be found in the Appendix section of this paper.

Regardless of this example's complexity, it is worth noting that it takes less than two minutes to develop, and 0.2 seconds to compile (with no link phase needed!). A similar example developed using any GUI builder would take much longer to assemble, and regardless of the development method used, compilation and linking using X windows is always very time consuming. An equivalent example written using Motif takes 20 seconds to compile and link. Similar ratios are seen for larger examples.

Although compilation time may not be the largest factor in a development effort, fast compiler turnaround reduces developer frustration, and increases productivity. A fast compiler and realization engine also encourages the exploration of alternatives, leading to a better final product.

In this example, the dialogue declares a single variable called `counter`, and uses the `display` agent to create four objects (one `canvas`, one `label`, and two `buttons`). The placement of the buttons and labels are specified in absolute coordinates, although the Appendix section shows more complex placement schemes involving relative placement with optional offsets.

The `text` attribute of the `show_it` button is defined to be dependent on the variable `counter`. This means that whenever the value of `counter` changes (in this example, by pushing the `click` button), the value of `show_it.text` is also changed. Thus, every time the `click` button is pushed, the `show_it` label is incremented by one. This dependency-based specification enables the dialogue writer to specify arbitrarily complex interrelationships between agents and attributes. The fact that the `text` attribute is a string and the value of the counter is irrelevant – the dialogue performs automatic type conversion.

```
use display;

VARIABLES:
   counter  : 0;

AGGREGATES:
draw_here : canvas {
   height   : 300;
   width    : 150;
   }

show_it : label {
   parent   : draw_here;
   x        : 50;
   y        : 50;
   text     : counter;
   }

click : button {
   parent   : draw_here;
   x        : 50;
   y        : 150;
   text     : "Push";
   pushed   : begin counter += 1 end
   }

quit : button {
   parent   : draw_here;
   x        : 50;
   y        : 200;
   text     : "Quit";
   pushed   : begin exit () end
   }
```

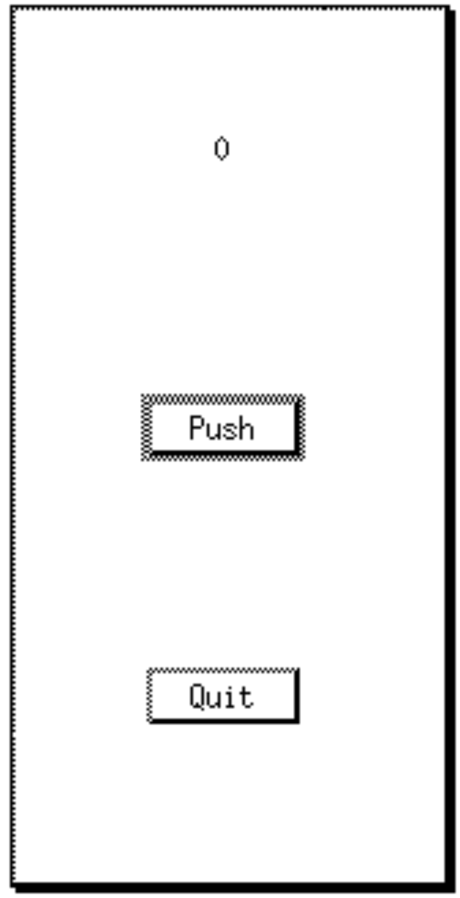

Figure 3 – A simple dialogue example

9. Conclusions

Using a UIMS can greatly speed application development and enhance maintainability. This is because the components of the systems – application, dialogue, and technology – are separated from each other. This means that:

1) Each component can be developed independently. Parallel development is faster than serial.

2) Each component can be developed by a person expert in that particular field. Graphics experts can develop technology agents, human factors experts can develop the dialogue, and application experts can develop the application agent.

3) Changes to one component of the system do not affect other components. The application agent may be changed independently of the dialogue, and the technology agent may be made to support new objects without changing the dialogue. This separation greatly reduces maintenance costs.

4) Overall development time on any component is greatly reduced. The dialogue complier is exceptionally fast, so many alternative dialogues can be tested before the final interface is selected.

5) Components can be reused. For example, one technology agent can be used with many systems. If different interfaces are desired, multiple dialogues can be used without changing the application agents. A single dialogue can also be told to use different technology agents, so a dialogue can display to X windows or Windows NT.

6) Upgrades to new versions of technologies are easier. When a new release of a toolkit is purchased, only the technology agents need to be recompiled (and not every dialogue using it). If one technology agent is shared (as is usual), once that agent is upgraded, *all* the dialogues using it are upgraded without recompilation or relinking.

7) The UIMS is inherently distributable, and can run on a heterogeneous network. Each of the components can run on a different machine, communicating over a network.

Appendix – Using a UIMS to Build Two Interfaces to One Application

Because the dialogue, technology, and application are separate, the UIMS easily allows an application to have multiple interfaces. This appendix section shows a simple application (a clock) with two radically different interfaces. The main components of the system (the interface definition, the dialogue, and the application agent) are shown.

A.1. `clock.idl`

This is the interface definition for the objects used by the clock agent. The executable image is defined to be `time_of_day`, and a single object type is defined here, namely a `daytime` object. The dialogue or the application agent may create instance of this object.

```
application objects <<"time_of_day">>

daytime {
   hrs       : integer;
   mins      : integer;
   secs      : integer;
   date      : string;
   }
```

A.2. `time_of_day.c`

This is the `time_of_day` agent for the clock interfaces Once the interface definition (in `clock.idl`) has been compiled, it creates the C language definitions of the objects, which can then be used by the agent and the dialogue.

Note that the agent does not actually change the time of day – it simply maintains an internal delta. Consequently, the agent is much more complex than really it needs to be. It would be simpler to just change the time of day with `settimeofday`, but we don’t assume root privileges.

```
#include "interface.h"                   /* The interface library */
#include "clock.h"                   /* The output from clock.idl */
#include <time.h>
#include <stdlib.h>

static daytime_obj *delta;      /* daytime_obj comes from clock.h */
static daytime_obj last_time;
```

```
static void find_time (daytime_obj *dto)
{
   time_t when;
   struct tm *now;

   when = time ((time_t *)0);
   now = localtime (&when);
   dto->hrs = now->tm_hour;
   dto->mins = now->tm_min;
   dto->secs = now->tm_sec;
   dto->date = malloc (20);
   (void) strptime (dto->date, "%B %e, %Y", now);
}

static void clock_create (xdl_idx obj_num, void *obj)
{
   delta = obj;
}

static void adjust_time (daytime_obj *dto)
{
   int hrs, mins, secs;
   /* If the delta hasn't been created yet, no adjustment needed */
   if (delta == (daytime_obj *)0)
      return;

   get_attr (&secs, delta, daytime$secs);
   if (dto->secs + secs < 0) {
      secs += 60;
      set_attr (delta, daytime$secs, secs);
      }
   else if (dto->secs + secs >= 60) {
      secs -= 60;
      set_attr (delta, daytime$secs, secs);
      }
   dto->secs += secs;

   get_attr (&mins, delta, daytime$mins);
   if (dto->mins + mins < 0) {
      mins += 60;
      set_attr (delta, daytime$mins, mins);
      }
   else if (dto->mins + mins >= 60) {
      mins -= 60;
      set_attr (delta, daytime$mins, mins);
      }
   dto->mins += mins;

   get_attr (&hrs, delta, daytime$hrs);
   if (dto->hrs + hrs < 0) {
      hrs += 24;
      set_attr (delta, daytime$hrs, hrs);
      }
   else if (dto->hrs + hrs >= 24) {
      hrs -= 24;
      set_attr (delta, daytime$hrs, hrs);
      }
   dto->hrs += hrs;
}
```

```
static void done (void)
{
   exit (0);
}

main (int argc, char **argv)
{
   daytime_obj *dlg_time;
   /*
    * This is a trivial router - it only processes create, modify and
    * dialogue shutdown actions.
    */
   static router_t daytime_router = {
      (a_none)0, clock_create, (a_modify_attr)0, (a_modify_row)0,
      (a_modify_idx)0, (a_destroy_object) 0, (a_begin_group) 0,
      (a_end_group) 0, done
      };

   Alpha_init ("clock.xdl");
   register_type_router (clock$daytime, &daytime_router);
   /* Create the daytime object for the interface to use */
   begin_group ();
   dlg_time = create_object (clock$daytime);
   find_time (&last_time);
   *dlg_time = last_time;                    /* Copy whole object */
   changed_object (dlg_time);
   end_group ();

   while (1) {
      sleep (100*MS);
      find_time (&last_time);                   /* Get time of day */
      while (change_available ())  /* See if the user has changes */
            process_next_change (TRUE);
      adjust_time (&last_time);      /* Adjust according to delta */
      begin_group ();
      *dlg_time = last_time;                 /* Copy whole object */
      changed_object (dlg_time);     /* Ship changes to interface */
      end_group ();
      }
}
```

A.3. `clock1.sl`

This dialogue creates an interface for a digital clock. The technology agent is called `display` (for the sake of brevity, it is not shown here). The time of day and date fields come from the application agent called `clock` (the name of the executable image is defined in `clock.idl`). The dialogue also accepts changes in the time of day via push-buttons, and sends those changes to the `clock` agent. Figure 4 shows the way the clock appears as various controls are interacted with.

Note that the hours, minutes, and seconds fields shown using the `display` agent are dependent on the values managed by the `clock` agent. Whenever the `clock` agent changes the values of the `daytime` object, the dialogue automatically instructs the `display` agent to make the corresponding changes on the screen. The `clock` agent has no idea how the date and time are being displayed (if at all!). Similarly, the dialogue does not know how the `display` agent actually gets the bits on the screen, nor how the `clock` agent determines the time (or how often). The dialogue simply expresses the interdependencies between objects in potentially disparate sections of the application, and routes information between them. In this way, the component parts are much simpler than if they knew details about the other parts of the system.

There are a number of specific features to note in the `clock1` dialogue (look for the cross reference numbers in the code):

❶ Two agents. In this example, we use a technology agent (`display`) to draw things on the screen and an application agent (`clock`) to maintain the time of day.

❷ Abstract object templates. An `xlabel` is the same type as a `label`, except the `parent` attribute is set to `drawing_area` (unless otherwise overridden). This enables dialogues to be written with less typing, and allows abstract objects to be created. These abstract objects can be substantially more complex (see item ❻).

❸ Font and appearance objects. Because many objects can share the same appearance, there is no reason to force the redeclaration of these appearances for each object. An appearance object allows many objects to have the same appearance, simply by referencing it. If the appearance object is changed, all objects which reference it also change in appearance (so all objects in a group can be made to change color or font by simply changing a single appearance object).

❹ Relative placement. An object can have its position dependent on the size of other objects – if the `drawing_area` were to change size, so would the position of the `quit` button. See item ❼ for another type of relative placement.

❺ Collector. A collector is a way of grouping a set of objects with a common existence condition. In this case, the existence is bound to the creation of a `daytime` object by the `clock` agent. When an instance of the object is created, an instance of the collector is created (which in turn causes all of the buttons and labels to be created). Another example of a collector is shown in item ❾.

❻ A more complex object abstraction – in this case the height and width, sensitivity and parent attributes are all set. Note that this means that the "colon" characters separating the hours, minutes and seconds are really tiny arrow buttons which are insensitive. When the `set_time` button is pushed in, the arrows grow and become sensitive, allowing the user to change the time.

❼ Relative placement. Although an object's placement can be specified in absolute coordinates, an object can also be said to be across or down from another object. In this case, the `hrs_down` button is to the right of the `hours` field and `down` from the `hrs_up` button. The `hrs_up` object (above) also adds a relative `y` offset of 7 units if the `set` attribute of the `set_time` button is off.

❽ Conditional formatting. The `seconds` label will also have "AM" or "PM" appended to it if we are in 12 hour mode (determined by the state of the `civilian` button).

❾ Another collector. This collector has as its existence condition the value of the `set` attribute of the `show_date` toggle button. If the button is "on", the label called `today` appears. If it is "off", the button disappears.

```
use display;                                                   ❶
use clock;

AGGREGATES :

drawing_area : canvas {
   height : 200;
   width : 400;
   }
```

```
xlabel => label {                                                  ❷
   parent => drawing_area;
   }

cour20 : font {                                                    ❸
   family : "courier";
   weight : "bold";
   slant : "r";
   pixel_size : 20;
   }

c20 : appearance {
   font : times20;
   }

cour34 : font {
   family : "courier";
   weight : "bold";
   slant : "r";
   pixel_size : 34;
   }

cb34 : appearance {
   font : cour34;
   }

quit : button {
   parent : drawing_area;
   text : "Off";
   height : 35;
   width : 40;
   x : drawing_area.width - quit.width - 10;                       ❹
   y : drawing_area.height - quit.height - 10;
   pushed : begin exit () end
   }

all_the_stuff : collector bound_to (daytime) {                     ❺
   AGGREGATES:
   one_of_these : exclusive_canvas {
      parent : drawing_area;
      height : 100;
      width : 200;
      x : 95;
      y : 65;
      }

   civilian : exclusive_button {
      parent : one_of_these;
      text : "12 Hour";
      }

   military : exclusive_button {
      parent : one_of_these;
      down : civilian;
      text : "24 Hour";
      set : true;
      }
```

```
set_show : toggle_canvas {
   parent : drawing_area;
   x : 200;
   y : 65;
   }

Set_Time : toggle_button {
   parent : set_show;
   text : "Set";
   }

Show_Date : toggle_button {
   parent : set_show;
   down : Set_Time;
   text : "Show Date";
   }

delta : daytime { /* Initialized to zero */ }

clock_area : canvas {
   parent : drawing_area;
   x : 75;
   y : 15;
   }

clabel => label {
   parent => clock_area;
   }
carrow_button => arrow_button {                                   ❻
   parent => clock_area;
   sensitive => Set_Time.set;
   height => Set_Time.set ? 18 : 8;
   width => Set_Time.set ? 18 : 8;
   }

hours : clabel {
   appearance : cb34;
   text : civilian.set ?
               daytime.hrs > 12 ?
                     int_to_str (daytime.hrs - 12, "%02d") :
                     int_to_str (daytime.hrs, "%02d") :
               int_to_str (daytime.hrs, "%02d");
   }

hrs_up : carrow_button {               /* Increases hours by 1 */
   direction : arrow_up;
   y : Set_Time.set ? 0 : 7;
   across : hours;
   pushed : begin delta.hrs += 1; end
   }

hrs_down : carrow_button {             /* Decreases hours by 1 */
   direction : arrow_down;
   across : hours;                                                ❼
   down : hrs_up;
   pushed : begin delta.hrs -= 1; end
   }
```

```
   minutes : clabel {
      appearance : cb34;
      text : int_to_str (daytime.mins, "%02d");
      across : hrs_up;
      }

   mins_up : carrow_button {
      direction : arrow_up;
      y : Set_Time.set ? 0 : 7;
      across : minutes;
      pushed : begin delta.mins += 1; end
      }

   mins_down : carrow_button {
      direction : arrow_down;
      across : minutes;
      down : mins_up;
      pushed : begin delta.mins -= 1; end
      }

   seconds : clabel {
      appearance : cb34;
      text : int_to_str(daytime.secs, "%02d") @                    ❽
            (civilian.set ? (daytime.hrs < 12 ? " AM" : " PM") : "");
      across : mins_up;
      }

   secs_set : collector exists if (Set_Time.set) endif {
      AGGREGATES:
      secs_up : carrow_button {
            direction : arrow_up;
            across : seconds;
            pushed : begin delta.secs += 1; end
            }

      secs_down : carrow_button {
            direction : arrow_down;
            across : seconds;
            down : secs_up;
            pushed : begin delta.secs -= 1; end
            }
      }

   todays_date : collector exists if Show_Date.set endif {         ❾
      AGGREGATES:
      today : xlabel {
            appearance : c20;
            text : daytime.date;
            x : 85;
            y : 130;
            }
      }
}
```

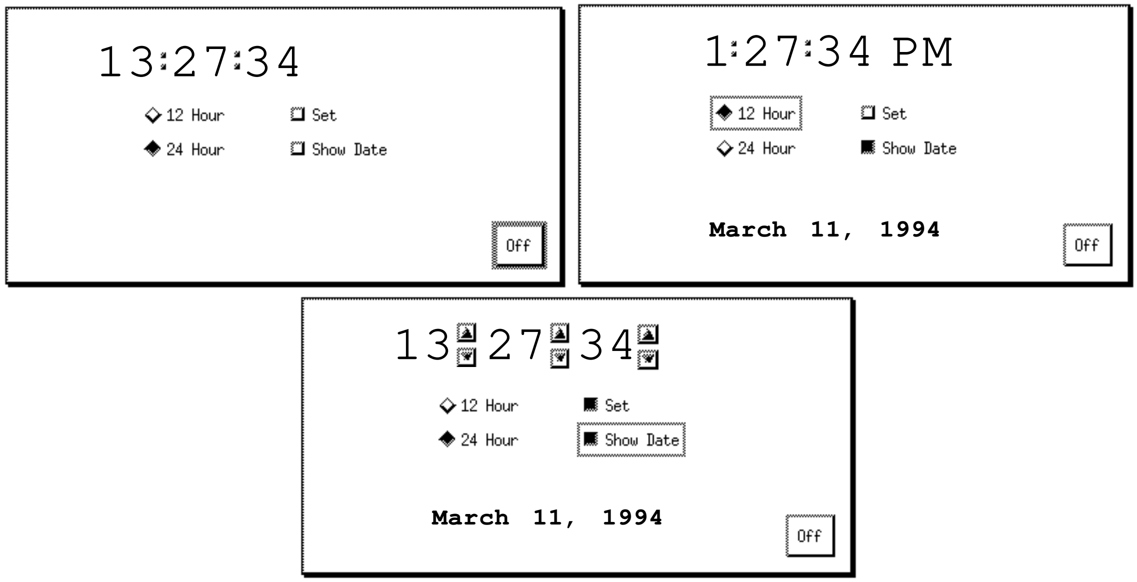

Figure 4 – Various appearances of `clock1.sl`

A.4. `clock2.sl`

This shows a different dialogue (or interface) for the same two agents as in the previous example. Neither agent (`display` or `clock`) has been changed (or recompiled, or even relinked!). Only the dialogue is different, giving a different look-and-feel to the same application.

In this second example, the clock is displayed as sliders – the time is changed by moving the slider bar, and the user does not see the date. Figure 5 shows the new interface defined with `clock2.sl`. This second example demonstrates that one application can be given multiple interfaces, depending on the functions the designer wishes to expose to the user. These differences in interaction can be completely invisible to the application agent.

```
use display;
use clock;

AGGREGATES :

drawing_area : canvas {
   height : 150;
   width : 250;
   }

quit : button {
   parent : drawing_area;
   text : "Off";
   height : 35;
   width : 40;
   x : drawing_area.width - quit.width - 10;
   y : drawing_area.height - quit.height - 10;
   pushed : begin exit () end
   }
```

```
all_the_stuff : collector bound_to (daytime) {
   AGGREGATES:

   delta : daytime { /* Initialized to zero */ }

   hours : slider {
      parent : drawing_area;
      y : 20;
      minimum : 1;
      maximum : 12;
      value : daytime.hrs mod 12 == 0 ? 12 : daytime.hrs mod 12;
      orientation : LeftToRight;
      changed : begin delta.hrs += hours.value - daytime.hrs; end
      }

   meridian : label {
      parent : drawing_area;
      across : hours;
      y : 20;
      text : daytime.hrs < 12 ? "AM" : "PM";
      }

   minutes : slider {
      parent : drawing_area;
      down : hours;
      maximum : 60;                     /* Minimum defaults to 0 */
      value : daytime.mins;
      orientation : LeftToRight;
      changed : begin delta.mins += minutes.value - daytime.mins; end
      }

   seconds : slider {
      parent : drawing_area;
      down : minutes;
      maximum : 60;                     /* Minimum defaults to 0 */
      value : daytime.secs;
      orientation : LeftToRight;
      changed : begin delta.secs += seconds.value - daytime.secs; end
      }
   }
```

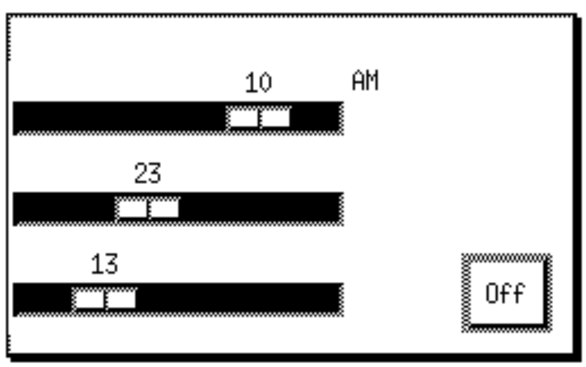

Figure 5 – Appearance of `clock2.sl`

Bibliography

[Arch92] The UIMS Tool Developers Workshop, "A Metamodel for the Runtime Architecture of an Interactive System," *SIGCHI Bulletin*, 24:1(32) Jan. 1992

[Bass91] Bass, Len; Joëlle Coutaz; *Developing Software for the User Interface*, Addison Wesley Publishing Company, 1991, ISBN 0-201-51046-4

[Bass90] Bass, Len, Brian Clapper, Erik Hardy, Rick Kazman, Robert Seacord; "Serpent: A User Interface Environment," *Proceedings, 1990 USENIX Winter Conference*, Jan. 1990

[Boehm88] Boehm, Barry W.; "A Spiral Model of Software Development and Enhancement," *IEEE Computer*, 21:5, May 1988

[Fisher87] Fisher, Gary E.; *Application Software Prototyping and Fourth Generation Languages*, National Bureau of Standards, May 1987

[Foley89] Foley, James, et al.; "Defining Interfaces at a High Level of Abstraction," *IEEE Software*, Jan. 1989

[Hardy91] Hardy, Erik J., Daniel V. Klein; "The Serpent UIMS", *Proceedings of the European Unix User's Group*, Nice FRANCE, Oct. 1990

[Kasik89] Kasik, David J., et al.; "Reflections on Using a UIMS for Complex Applications," *IEEE Software*, Jan. 1989

[Klein91] Klein, D.V.; *Serpent: System Guide*, Software Engineering Institute User's Guide CMU/SEI-91-UG-2, April 1991

[Kolodziej87] Kolodziej, Stan; "User Interface Management Systems," *Computerworld*, July 8, 1987

[Myers88a] Myers, Brad A.; *Tools for Creating User Interfaces: an Introduction and Survey,* Carnegie Mellon University, 1988, CMU-CS-88-107

[Myers88b] Myers, Brad A.; *The Garnet User Interface Development Environment: a Proposal*, Carnegie Mellon University, 1988, CMU-CS-88-153

[Pfaff85] Pfaff, G. (Ed.); *Seeheim Workshop on User Interface Management Systems*, Springer-Verlag, Berlin, 1985