The Amulet Prototype-Instance Framework

Brad A. Myers, Richard G. McDaniel, and Robert C. Miller

December 22, 1998

To appear in:
Object-Oriented Application Frameworks, vol. 3,
edited by Mohamed Fayad and Douglas C. Schmidt.
New York: John Wiley & Sons, 1999.

Human Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891

(412) 268-5150
FAX: (412) 268-1266
bam@cs.cmu.edu
http://www.cs.cmu.edu/~amulet

Abstract

Amulet is a new kind of object-oriented framework for user interface development that is based on a *prototype-instance* object system instead of the conventional class-instance object system. In a prototype-instance object system, there is no concept of a "class" since every object can serve as a prototype for other objects, and any instance can override any methods or data values. Amulet is also differentiated by high-level encapsulations of interactive behaviors, and by the ubiquitous use of *constraints*, which are relationships that the programmer declares once and then are enforced by the system. The result is that programs written using the Amulet framework have a different style than those written with conventional frameworks. For instance, Amulet applications are typically constructed by combining instances of the built-in objects, rather than by subclassing the built-in classes or writing methods. Amulet is written in C++ and is portable across Windows NT and 95, Unix X/11, and the Macintosh.

This is a revision of:
Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, and Patrick Doane "The Amulet Environment: New Models for Effective User Interface Software Development," *IEEE Transactions on Software Engineering*. vol. 23, no. 6. June, 1997. pp. 347-365.

This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NCCOSC or the U.S. Government.

Creating user interface software has proven to be very difficult and expensive, because it is often large and complex, and challenging to implement, debug, and modify. Most of today's application frameworks for user interfaces still leave far too much of the application to the programmer. The Amulet user interface development framework tries to overcome this problem by supplying high-level support for the *insides* of application windows, which most other frameworks ignore. For example, whereas most frameworks do a good job of managing the creation of windows and the main menus, they leave the *contents* of the window to be programmed at the low-level window manager level, accepting events like "mouse left-button down at 30,50" and using routines like "draw-line." In contrast, Amulet supplies high-level support for the graphics and interactive behaviors of application-specific objects. The result is that many behaviors, such as creating, moving, selecting, and manipulating objects, cut/copy/paste, save and load, undoing of operations, etc., can often be incorporated into applications without writing any methods at all.

A key reason that Amulet provides a high level of support is that all of the user interface objects are available at run-time for inspection and manipulation through a standard protocol. This allows high-level, built-in utilities to be provided which, in other toolkits and frameworks, must be re-implemented for each application. For example, the graphical selection handles widget can get the list of objects in a window, find out which ones are graphical, and move and resize a selected object, all using standard protocols, even if the objects are custom-created and application-specific.

The result is that creating applications in Amulet is quite different than in other frameworks. In fact, much of Amulet programming is done without writing methods (Myers 1992), but instead by creating instances of built-in objects, setting their properties, and combining them into groups.

In addition to incorporating innovations into its own design, Amulet has an open architecture to enable user interface researchers and developers to easily investigate their own innovations. For example, Amulet is the first system that supports multiple constraint *solvers* operating at the same time, so that researchers might be able to investigate new kinds of constraint solvers. The undo model also supports new designs. The "widgets" (the UNIX name for elements of the toolkit, like scrollbars, buttons and menus; sometimes called "controls" on the PC) are implemented in an open fashion using the Amulet intrinsics so that researchers can replace or modify the widgets. The goal is that researchers will only have to implement the parts that they are interested in, relying on the Amulet library for everything else. In addition, we aim for Amulet to be useful for students and general developers. Therefore, we have tried to make Amulet easy to learn, and to have sufficient robustness, performance and documentation to attract a wide audience.

Amulet, which stands for Automatic Manufacture of Usable and Learnable Editors and Toolkits (Myers 1997), is being developed as a research project at Carnegie Mellon University. It is implemented in C++ and runs on X/11, Windows 95, Windows NT, and the Macintosh. Applications created using Amulet can simply be recompiled to run on any of the machines.

1. Layered Design

The Amulet framework is divided into a number of layers (see Figure 1). These layers include an abstract interface to the window managers; and novel models for objects, constraints, input, output, and commands; and a set of widgets. The following sections describe the overall design of each of these.

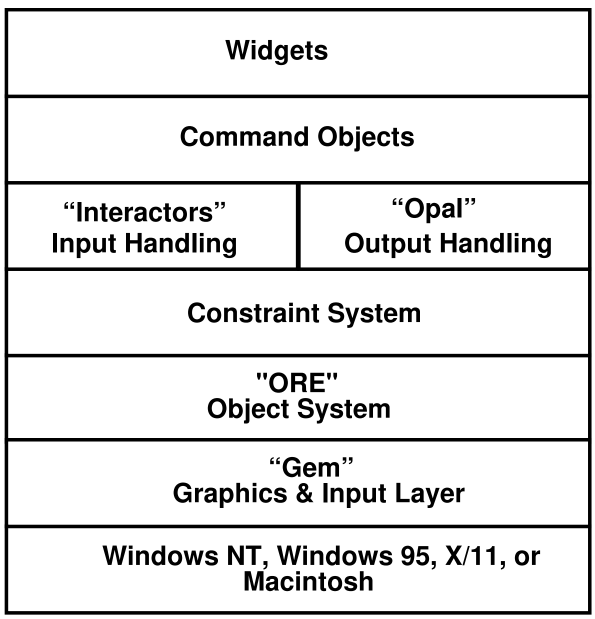


Figure 1: The overall structure of the Amulet system.

1.1. Gem: Abstract Interface to the Window Managers

Amulet provides a portable interface to various window managers called “Gem,” which stands for the Graphics and Events Manager. Gem uses C++ mechanisms to provide a simple graphics and input interface used by the rest of Amulet. Any code written using Gem will port to different windowing systems (Windows 95 or NT, Macintosh, or X/11) without change. Most other toolkits and frameworks only provide an interface at the Gem level, or require that all graphics use the underlying low-level window manager drawing interface. However, typical Amulet users never see the Gem interface, since the higher-level parts of the Amulet framework provide access to the same capabilities in an easier-to-use way. We export the Gem interface mainly for advanced Amulet users. If the programmer wants to make something very efficient, calling Gem directly may be appropriate.

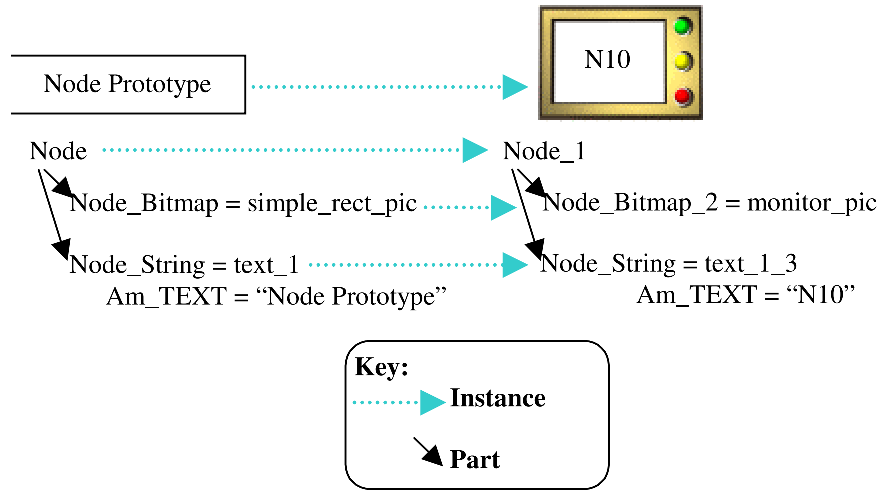


Figure 2: When an instance is made of the node prototype, an instance is also made of each of the parts.

1.2. Object System

The “Ore” (Object Registering and Encoding) layer of Amulet implements a *prototype-instance* object and constraint system (Lieberman 1986) on top of C++. In Amulet’s prototype-instance object system, there is no concept of a “class” since every object can serve as a prototype for other objects. An object is comprised of a set of *slots*. A “slot” has a name and can hold a value of any type, for example the slot named `Am_LEFT` might hold the value 10. Slots are similar to member variables or instance variables in other object systems. A new object is created by making an *instance* of another object, which is called the *prototype*. Creating an instance is like making a copy of an object, except for the way that inheritance works. One of the innovations in Amulet is that if an object has parts, then the instance will also get instances of the parts (see Figure 2). We call this “structural inheritance.”

An instance starts off inheriting all of its slot values from the prototype, so that initially the instance and its prototype have all the same values. Then, the programmer will typically set some slots of the instance with new values. If a slot is *not* set in the instance, then its value will change if the prototype’s value is later changed. The object system is *dynamic* in that slots in objects can be added and removed from objects at run time, and the types in slots can also change.

For example, the following Amulet code creates a `Node_Bitmap` as an instance of the built-in `Am_Bitmap` object, and then sets the `Am_IMAGE` slot to the appropriate picture.

```
Am_Object Node_Bitmap = Am_Bitmap.Create()
                            .Set(Am_IMAGE, simple_rect_pic);
```

There is nothing special about the objects in the library (like the `Am_Bitmap` object used above). Any object can serve as a prototype from which to create other objects. For example, the following code creates an instance of the `Node_Bitmap`, changes the picture, and then puts it in a particular place. Other slots that are not set, such as the `Am_VISIBLE` slot, retain their default, inherited value.

```
Am_Object Node_Bitmap_2 = Node_Bitmap.Create()
                              .Set(Am_IMAGE, monitor_pic)
                              .Set(Am_LEFT, 10)
                              .Set(Am_TOP, 43);
```

An important feature of Amulet's object system is that there is no distinction between *methods* and *data*: any instance can override an inherited method as easily as inherited data. In a conventional class-instance model such as Smalltalk or C++, instances can have different data, but only sub-classes can have different methods. Thus, in cases where each instance needs a unique method, conventional systems must use a mechanism other than the regular method invocation, or create a new subclass and a single instance of that subclass each time. For example, a button widget might use a regular C++ method for drawing, but would have to use a different mechanism for the call-back procedure used when the user clicks on the button, since each instance of the button needs a different call-back. In Amulet, the draw method and the callback use the same mechanism.

As an example, the following sets the "`DO`" method of the command in a button to be `my_method` (command objects are described below). Then, `my_method` will be called whenever the user hits the button.

```
my_button.Get_Object(Am_COMMAND).Set(Am_DO_METHOD, my_method);
```

Amulet's object system also contains many other features that may be useful for programmers. An automatic memory management mechanism, which uses a reference counting scheme, manages Amulet's objects. A complete set of querying functions makes it easy to determine objects' properties at run-time. These are used by the debugging facilities described below, and they can also be useful for application programs. For example, a program can query the list of slots of an object, and whether each slot is local or inherited. For each slot, the name of the slot (e.g. `"LEFT"`), current value (e.g., `10`), and type of the current value (e.g., `Am_INT_TYPE`) can be retrieved. Since there is no distinction between method and data slots, this one mechanism is used to discover all of the properties of an object.

Programming with a prototype-instance object system is a quite different style than is used in conventional object-oriented languages. Much of the code is devoted to listing the slots and default values for prototype objects, usually at initialization time, and then creating instances, possibly overriding some slots, as the program is running. This “declarative style” of programming seems to be intuitive and less error prone, and it makes it easier for more of the code to be created and analyzed by interactive design tools, like interface builders.

There are many other advantages of the prototype-instance model. Having no distinction between classes and instances, or between methods and data, means that there are fewer concepts for the programmer to learn and a consistent mechanism can be used everywhere. Another advantage of the prototype-instance object system is that it is very dynamic and flexible. All of the properties of objects can be set and queried at run time, and interactive tools can easily read and set these properties. In fact, most of today’s frameworks and toolkits implement some form of “attribute-value pairs” to hold the properties of the widgets, but Amulet’s object system provides significantly more flexibility and capabilities.

Another advantage of our prototype-instance object systems over C++ is the ability to treat “classes” as first-class objects. In C++, one cannot store a class object in a variable so that different kinds of objects can be created at run-time. For example, C++ does not allow code like:

```
obj_to_create = Rectangle;        //NOT ALLOWED IN C++
…
new_object = new obj_to_create;   //NOT ALLOWED IN C++
```

Instead, the operand of the `new` operator must be a fixed class, leading to large `case` statements and other inflexible and error-prone constructs. In contrast, since one can create an instance of any object in a prototype-instance object system, one can store a reference to an object in a variable and later use the variable to create a new object:

```
Am_Object obj_to_create = Am_Rectangle; //Typical Amulet code
…
Am_Object new_object = obj_to_create.Create();
```

Amulet’s predecessor, Garnet, also used a prototype-instance object system (Myers 1992). Amulet’s design is more complete and flexible, and we fixed a number of problems we experienced with Garnet. For example, we added the ability for a programmer to declare that certain slots are *not* inherited, so that each instance will start out with no value for that slot. For example, the `Am_Drawonable` slot of a `Window` object holds the machine-specific pointer to the underlying window-manager window, and should not be inherited by instances of the window. Each instance of a `Window` must create and assign its own value for the `Am_Drawonable` slot. Finally, it is worth pointing out that Amulet is able to provide dynamic slot typing, a dynamic prototype-instance system, and constraints in C++ without using a preprocessor or a scripting language.

Although designed to support the creation of graphical objects, many Amulet users have discovered that the prototype-instance object system is useful for representing their internal application data whenever flexible and dynamically changing data types are desired.

The main disadvantage of the prototype-instance model over the conventional class-instance model is performance. When a slot is accessed, the system must perform a search through the object to see if the slot is there, and if not, it must search the prototypes up to the root. The same search is needed for both method and data slots. We have investigated various indexing and hashing schemes to help reduce this overhead, including hardwiring some common slots, but this increases complexity and sometimes does not improve performance. Dynamic type checking also adds some overhead. The forward and backward pointers and space for the types add space overhead. The Self prototype-instance system (Ungar 1987) uses extensive compiler techniques to try to remove some of this search, but we have not found this necessary to achieve adequate performance on modern platforms. One reason that Self needs this is that it uses the prototype-instance system for *everything*, right down to integer arithmetic, whereas Amulet uses the efficient underlying C and C++ mechanisms for basic computation and only uses the prototype-instance model for the user interface.

Another disadvantage of our dynamic prototype-instance model is the lack of compile-time checking. Instead, Amulet uses run-time type-checking. However, our experience over the last 10 years is that very few bugs are caused by type errors found at run-time, and the added flexibility far outweighs the problems.

1.3. Constraints

Amulet integrates *constraint solving* with the object system. This means that instead of containing a constant value like a number or a string, any slot of any object can contain an expression which computes the value. If the expression refers to slots of other objects, then when those objects are changed, the expression is automatically re-evaluated. This kind of constraint resembles a spreadsheet formula, so it is called a “formula constraint” in Amulet. Constraint expressions can contain arbitrary C++ code. This works because the standard "Get" method that accesses values of slots can tell whether it is being invoked from inside of a formula, and if so, in addition to returning the value, it also sets up a dependency. Then, whenever a slot's value changes, Amulet knows which constraints depend on that value, and can cause the constraints to recalculate.

Although many other research systems have provided constraints, we were the first to truly integrate them with the object system and make them general purpose. An important result of this is that constraints are used throughout the system in many different ways. For example, the built-in `text` object has constraints in its width and height slots that compute the dimensions based on the current string and font.

Amulet does not use a special preprocessor, so the syntax for specifying constraints is somewhat verbose. In C++, it is impossible to create new functions inside of other functions, so all formulas must be defined at the top level before they are used. For example:

```
// define a formula called right_of_tool_panel_formula which returns an int
Am_Define_Formula(int, right_of_tool_panel_formula) {
  // 5 pixels away from the right of the tool_panel
  return (int)tool_panel.Get(Am_LEFT) +
            (int)tool_panel.Get(Am_WIDTH) + 5;
}
...
// now use the formula to compute the left of the scrolling_window
scrolling_window.Set(Am_LEFT, right_of_tool_panel_formula);
```

The macro `Am_Define_Formula` creates a formula named with the second argument (here `right_of_tool_panel_formula`) which returns the type of its first argument (here `int`). The `Am_Define_Formula` macro defines a procedure to be executed by the formula, and the code following the macro is used as the procedure's body. The formula contains a pointer to the procedure to execute, the name of the constraint for debugging and tracing, and the list of slots used by this constraint.

Slots are accessed the same way whether they contain constraints or constant values, and the user's code does not know how the value was calculated. Furthermore, constraints can be used for computing any type of value, not just integers for layout. For example, the label shown in a button widget can contain a constraint to choose the label based some property of an object. The following toggles the button's label based on the `Am_VALUE` slot of the object `other_obj`.

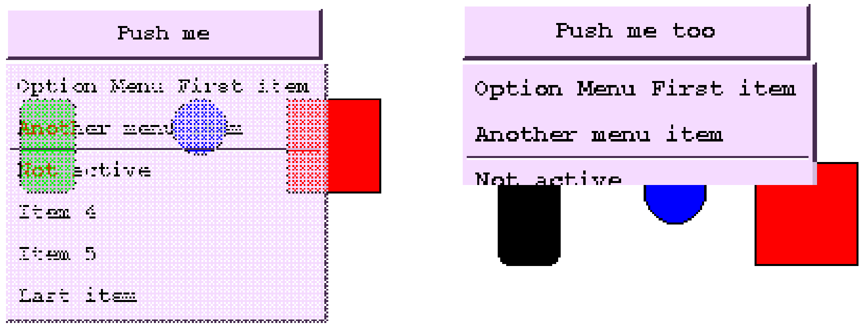
```
Am_Define_Formula(char*, compute_label_formula) {
    if (other_obj.Get(Am_VALUE) == true) return "Turn off value";
    else return "Turn on value";
}
my_button.Set(Am_LABEL, compute_label_formula);
```

The button widget does not care that the string was computed with a constraint. Whenever the value of a slot of an object changes, either because the programmer explicitly set the slot or because it contains a constraint which was automatically recalculated, the object will be redrawn if necessary, so here the button will be redrawn whenever `other_obj`'s `Am_VALUE` slot changes.

Our constraint system was the first to allow the dynamic computation of the objects to which a constraint refers, so a constraint can not only compute the value to return, but also *which objects* and slots to reference. This allows such constraints as "the width is the maximum of all the components" which will be updated whenever components are added or removed as well as when one of the components' position changes. Most other constraint systems cannot handle these kinds of constraints. These "indirect constraints" (Vander Zanden 1994) are also important for supporting object inheritance. When an instance is created of an object, Amulet also creates instances of any constraints in that object. These constraints refer to other objects indirectly. This is a form of "procedural abstraction" since the constraints can be considered as relationships that can be reused in multiple places. In fact, Amulet supplies a library of predefined constraints, which can be used for many of the basic relationships found frequently in user interfaces.

An important research area in user interface software is creating new kinds of constraint *solvers* (e.g. (Hudson 1996)). Therefore, Amulet contains an architecture that allows multiple solvers to co-exist. Currently, in addition to the one-way formula solver described above, Amulet supports a multi-output, multi-way solver, and an *animation* constraint solver (Myers 1996b). The animation constraint solver allows values of objects to change smoothly through time. Instead of jumping to a new value, the animation constraint forces the slot to take on a series of values interpolated between the old and new values. Animation constraints can work on any type of value, interpolating numbers, colors, or the point-list of a polygon. Special animation constraints are available for visibility, to cause an object to fade in and out, or grow and shrink (see Figure 3).

There is an important distinction between the style of programming supported by constraints versus that of conventional approaches. In a conventional framework, every place where an object might be changed must also know about the other things that might be affected by that change. For example, suppose the nodes of Figure 2 had lines pointing to them. Then, every place that could move or change the size of the nodes would also have to know about adjusting the lines so they stay attached. In a conventional object-oriented framework, this might be programmed by having the move and grow methods of the nodes know about the attached lines. In a non-object-oriented system like Visual Basic, code would have to be added to every place where the position or size of a node was set to deal with the lines. In both styles, if a new kind of object also wants to be informed of the node's position, the node would have to be reprogrammed to know about these new objects as well.



(a) (b)

Figure 3: Animation constraints can be added to the `Am_VISIBLE` slot of a popup menu to make it (a) fade in using halftoning or (b) grow from the top.

In contrast, in a constraint system, the *lines themselves* know how they are attached to the node, so the information is localized at the appropriate place. If some new object wants to stay attached to the node, a constraint can be put into the new object itself, and the code for the node does not have to be changed. Thus, constraints tend to more successfully modularize an application's code because the information about which objects depend on values is maintained automatically by the system, and does not need to be recoded into each manipulator of the value.

1.4. Opal Output Model

The graphical object layer of Amulet is called Opal, the Object Programming Aggregate Layer. Opal hides the graphics part of Gem and provides a convenient interface to the programmer by using a retained object model, also called a structured graphics model or a display list. The programmer creates instances of the built-in graphical object prototypes, like rectangles, lines, text, circles, and polygons, and adds them to a group or window. Then Amulet automatically redraws the appropriate parts of the window if it becomes uncovered, or if any properties of the objects change. This frees the programmer from having to deal with refresh. Objects can simply be created and deleted and their properties can be set. Furthermore, Opal automatically handles object creation and layout when the data can be displayed as lists or tables.

Opal makes heavy use of the object and constraint models of Amulet. Of course, all graphical objects are Amulet objects. The properties of objects that programmers do not care about can simply be ignored because they will inherit appropriate default values from prototypes. Groups use the ORE-level "structural inheritance" so that the parts of prototypes are created in instances (see Figure 2). This means that the programmer can simply create instances or copies of groups in the same way as primitive objects, and in many cases will not even know if an object is a primitive or a group. For example, some of the widgets are implemented as a group of objects, and others as a single object with a custom draw method. Due to this integration and uniform structure, simple programs are quite short and there are fewer concepts and features to learn. For example, *any kind* of object (even a group of objects) can be added as the label of a button or menu (not just strings or bitmaps as in Motif).

The retained object model allows Amulet to provide many facilities that must be programmed by each application in other frameworks and toolkits, including automatic refresh. Amulet uses a relatively efficient algorithm that calculates which objects will be affected when the object is erased, and redraws only the affected objects from back to front. Double buffering is provided by Amulet to eliminate flicker.

As an example, the following is the complete “hello world” program in Amulet, that displays a string, and redraws the string if the window becomes covered and then uncovered. This program would be about 2 pages long in Motif.

```
#include <amulet.h>
void main (void) {
  Am_Initialize ();   //initialize Amulet
  Am_Screen
    .Add_Part ( Am_Window.Create () //add a window to the screen using all of the default values
      .Add_Part ( Am_Text.Create ()   //create a text object and add it to the window
        .Set ( Am_TEXT, "Hello World!"))); //set the string
  Am_Main_Event_Loop ();   //display the window and then handle all input events
  Am_Cleanup (); //clean up Amulet
}
```

1.5. Interactors

Programming interactive behaviors has always been the hardest part of creating user interface software, especially since many frameworks only provide a stream of raw input events for each window which the programmer must interpret and manage. To solve this problem, we introduced the “Interactor” model for handling input, where each Interactor object type implements a particular kind of interactive behavior, such as moving an object with the mouse, or selecting one of a set of objects. To make a graphical object respond to input, the programmer simply attaches an instance of the appropriate type of Interactor to the graphics. The graphical object itself does not handle input events.

The Interactor design is one of the first to successfully separate the “Controller” from the “View” in the “Model-View-Controller” idea from Smalltalk (Krasner 1988). The model contains the data, the view presents the data, and the controller manipulates the view. Most previous systems, including the original Smalltalk implementation, had the View and Controller tightly linked, in that the controller would have to be reimplemented whenever the view was changed, and vice versa. Indeed, many later systems such as Andrew (Palay 1988) and InterViews (Linton 1989) combined the view and controller and called both the "View." In contrast, Amulet's Interactors are independent of graphics, and can be reused in many different contexts.

Internally, each Interactor operates similarly. It waits for a particular starting event over a particular object or over any of a set of objects. For example, an Interactor to move the nodes of Figure 2 might wait for a left mouse button down over any of the nodes. When that event is seen, the Interactor starts running on the particular object clicked on, processing certain events. The moving Interactor processes mouse move events, while looking for a left button up event, or an abort event (usually Control-G, Command-dot, or ESC). While the Interactor is running, the user is supplied feedback, either as a separate object (such as a dotted rectangle following the mouse), or by having the original object itself move. If the Interactor is aborted because the user hits the appropriate key or by a program calling the abort method, the original object is restored to its original state, the feedback object is hidden, and the Interactor goes back to waiting for a start event. If the Interactor completes normally (because the mouse button was released), then the feedback is hidden, the graphical object is updated appropriately, and a "command object" is allocated (see below). Interactors are highly parameterized so that the programmer can specify the start, end, and abort events, the objects the Interactor operates over and uses for feedback, along with other aspects such as gridding and how many objects can be selected. As a result, Amulet's six types of Interactors are sufficient to cover all the behaviors found in today's interfaces. Evidence for this claim is that in *none* of the applications that have been created so far with Amulet, did programmers ever need to go around the Interactors to get to the underlying window manager events.

The six types of Interactors in Amulet are: *Choice Interactor*, which is used to choose one or more objects from a set; *One Shot Interactor*, which is used to cause something to happen immediately when an event occurs; *Move-Grow Interactor*, which is used to have a graphical object move or change size with the mouse; *New Points Interactor*, which is used to enter new points, such as when creating new objects; *Text Edit Interactor*, which supports editing the text string of a text object; and *Gesture Interactor*, which supports free-hand gestures, such as drawing an "X" over an object to delete it, or encircling a set of objects to select them.

The Interactors are implemented using Amulet objects, so parameters are simply slots the programmer can set, or leave at their default values. Constraints can also be used to compute the parameters. For example, the `Am_ACTIVE` slot of an Interactor often contains a constraint depending on the global mode, and a constraint in a single Move-Grow Interactor might determine whether objects are moved or grown based on which mouse button was held down.

Normally, the Interactor operates on the object to which it is attached. An important feature of Amulet's Interactors is that they can also operate on a *set* of objects. For example, the Choice Interactor can select among the elements of a group, and the Move-Grow Interactor can be attached to a window to manipulate any object added as a part of the window. By default, Interactors make this choice based on the type of object they are attached to (group versus non-group), but the programmer can explicitly specify which is desired.

As an example of the use of Interactors, the following code might be used to create new lines. Note that the constraint `line_tool_is_selected` is used to make this behavior be available only when the correct tool in the palette is selected.

```
all_objs.Add_Part (Am_New_Points_Interactor.Create("create_line")
                .Set(Am_AS_LINE, true)  //want to create a new line
                .Set(Am_FEEDBACK_OBJECT, lfeedback) //feedback while dragging
                .Set(Am_CREATE_NEW_OBJECT_METHOD, create_new_line)
                .Set(Am_ACTIVE, line_tool_is_selected))
```

The following are some examples that show how Interactors can be used for moving and selecting objects. Note that in the simplest cases, an object can be made interactive with a single line of code:

```
//allow my_object to be moved while the left mouse button is held down
my_object.Add_Part(Am_Move_Grow_Interactor.Create());

//allow any part added to my_group to be grown using the right button
my_group.Add_Part(Am_Move_Grow_Interactor.Create()
                    .Set(Am_GROWING, true)
                    .Set(Am_START_EVENT, "right_down"));

//allow one or more parts of my_group to be selected with the left button
my_group.Add_Part(Am_Choice_Interactor.Create()
                    .Set(Am_HOW_SET, Am_CHOICE_LIST_TOGGLE));
```

A common design in other frameworks is to just have each graphical object have a standard set of methods for the events that it handles. Visual Basic is an example of this design, where the programmer can write methods that are activated when the user clicks on or drags an object. There are a number of advantages to Amulet's design of having *explicit* objects (the Interactors) to represent the behaviors of the graphics. First, it provides significantly greater reuse for such common features as gridding, undo, and enabling and disabling operations, since these are provided in a single place, instead of being re-implemented with each graphical object. Second, being able to analyze, inspect, and manipulate the behavior objects makes debugging and tracing easier, and enables external agents, tutors and alternative interfaces like speech and gestures to control the interface without modifying the graphical objects or the existing behavior logic.

1.6. Widgets

Amulet supplies a large set of widgets, including pull-down menus, buttons, check boxes, radio buttons, text-input fields, scroll bars, etc. Each widget has a different drawing routine for the Motif, Microsoft Windows, and the Macintosh look and feel. By default, the widgets will appear appropriately to the native machine, but the programmer can switch to any look on any machine (for testing). Amulet reimplements all the widgets rather than using the built-in widgets from the various toolkits so that we can provide flexibility and control to programmers who want to investigate new widget behaviors. This is necessary, for example, to create a scroll bar with two handles or to support multiple people operating with a widget at the same time for a multi-user application. Widgets are completely integrated with the object, constraint and command models, so properties of widgets can be computed by constraints, and the actions of widgets are represented by command objects (see next section), so they are easily undone. The various kinds of button and menu widgets can accept arbitrary Amulet objects including strings and bitmaps to display as the labels. This is easy in Amulet since there is a standard way for the buttons to query objects for their size and tell them where to draw.

Since widgets are objects, constraints are often used to compute their parameters, for example to enable and disable the widgets based on the current global state, and to lay out the widgets based on the window's current size. Amulet's widgets also have an extra parameter to disable them without graying out, which was added so the actual widgets can be used by interface-builder programs that need the widgets to be selected and moved when clicked on, instead of performing their normal functions.

In addition, Amulet supplies other widgets for the *insides* of application programs. For example, the selections-handles widget implements the familiar squares around the edges of graphical objects that show what is selected and allows the selected objects to be moved and resized. All other toolkits require programmers to re-implement selection handles and all their standard behaviors in every application, but in Amulet, programmers only need to add an instance of this widget to their window.

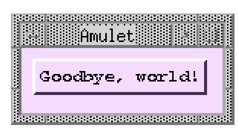
There are also built-in dialog boxes, like message and error boxes, but mostly dialog boxes are just ordinary Amulet windows that are made visible, and then made invisible (by simply setting the `Am_VISIBLE` slot of the window). Often, dialog boxes will be designed using Amulet's interface builder, called "Gilt," which stands for Graphical Interface Layout Tool. Gilt allows widgets to be laid out interactively using the mouse, and can set their properties.

1.7. Command Objects

Often, the Interactors and widgets operate simply by setting the appropriate slots of objects and having the values computed by constraints. In other cases, extra actions are required. Rather than using a "call-back procedure" (which is an application-specific procedure attached to a widget and called when the widget is clicked on by the user), Amulet allocates a *command object* and calls its "DO" method (Myers 1996a). Amulet's command objects also provide slots and methods to handle undo, selective undo and repeat, enabling and disabling the command (graying it out), help, and "balloon help" messages. Thus, unlike in Apple's MacApp framework (where the information about when a command is available is not programmed as a method of the command), the command objects in Amulet provide a single place for describing a behavior.

Furthermore, the command object architecture promotes reuse because commands for such high-level behaviors such as move-object, create-object, change-property, become-selected, cut, copy, paste, duplicate, quit, to-top and bottom, group and ungroup, undo and redo, and drag-and-drop are supplied in a library and can often be used by applications *without change.* This is possible because the retained object model means that there is a standard way to access and manipulate even application-specific objects. As a simple example, the following creates a button that uses the built-in Quit command to exit the program when the button is pressed.

```
Am_Object my_button = Am_Button.Create ()
      .Add_Part(Am_COMMAND, Am_Quit_Command.Create()
                 .Set (Am_LABEL, "Goodbye, world!"));
```



The commands in Amulet are *hierarchical*, so that a behavior may be composed of high-level and low-level commands (Myers 1996a). For example, a scroll bar command might internally use a move-object command. This improves modularity and reuse because each command is limited to its own local actions.

All of the built-in operations in Amulet support undo. Thus, if a programmer uses the standard Interactors and command objects, all operations are automatically undoable without writing any extra code. If the programmer creates custom commands that perform application-specific actions, then a custom undo method will have to be written as well. However, we have found that the Amulet object and constraint models make writing undo methods very easy since any needed data can be stored as slots in the command objects, and due to constraints, undoing operations is usually only a matter of resetting some slots. For example, the built-in command for moving objects simply saves the old position of the graphical object as the `Am_OLD_VALUE` slot of the command, so to undo the operation, the old position can simply be set into the slots of the graphical object. The graphical object itself does not need to have any methods or saved state to support undo.

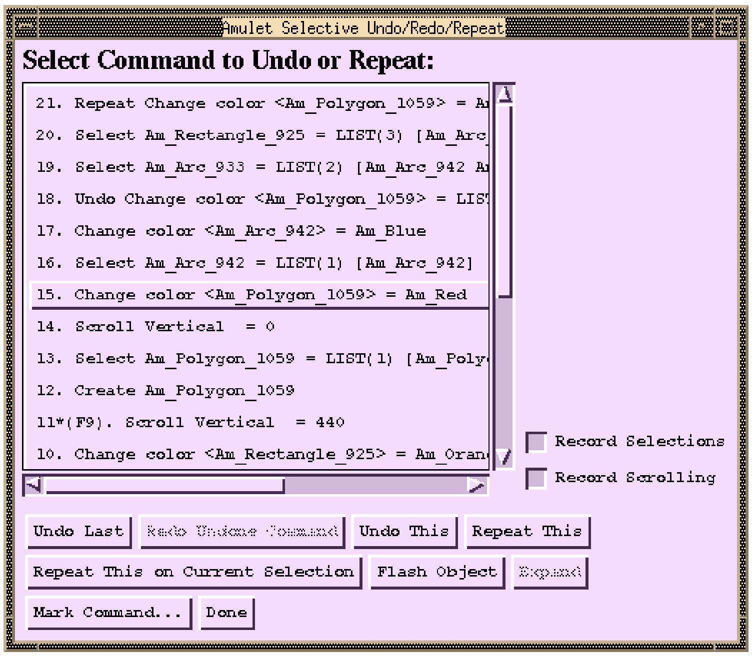


Figure 4: The experimental dialog box that allows users to access the regular undo and redo operations (the first two buttons below the scrolling list). Other buttons operate on the command selected in the list (here, number 15), and will undo it, repeat it, or repeat it on new objects. "Flash Object" shows the object associated with the command. The "Expand" button will allow a command that operates on multiple objects to be separated into separate commands. The display of each command shows the action, the name of the objects affected, and the new value. The radio buttons on the right cause scrolling and selection commands to be queued, so that, for example, an accidental deselection of a set of objects can be undone.

Amulet's commands also support investigation of various undo mechanisms. Currently, Amulet supplies three different undo mechanisms from which the developer can choose: single undo like the Macintosh, multiple undo like Microsoft Word Version 6 and Emacs, and a novel form of undo and repeat, where any previous command, including scrolling and selections, can be selectively undone, repeated on the *same* object, or repeated on a new selection (Myers 1996a). Figure 4 shows the experimental dialog box that is supplied to support this new style and allow users to select a previously executed command. Researchers can also create their own undo mechanism and integrate it into the Amulet system.

Amulet also supports scripting by generalizing a sequence of recorded commands (Myers 1998). The user can select some commands in Figure 4 for a script, and then give it a name. Parameters of the commands can be automatically or explicitly generalized so the script will work in various contexts. Scripts can be edited and the user can also specify various ways they can be invoked.

2. Outline of Typical Applications

Due to Amulet's prototype-instance object system, Interactors, and constraints, writing applications for Amulet uses a quite different style than with other object-oriented frameworks (Myers 1992). The first step is usually to create a set of prototype objects that will be used by the application, such as, for example, the `Node` of Figure 2. Next, a main window is created and added to the screen, as shown in the code for the "Hello World" example. Next, the programmer will usually add the necessary widgets and sub-windows to the window. Various slots for all these objects might be computed by constraints. Any dialog boxes or other secondary windows would also be created in the same way as the main windows, but their `Am_VISIBLE` slots will be set to `false` until they are needed. Finally, the Interactors and commands will be attached to objects. If necessary, custom `DO` and `UNDO` methods might be written for the command objects. However, unlike other frameworks, methods for drawing, saving, and printing are unnecessary due to the structured graphics model. Methods for the common interactive behaviors are also not necessary due to the Interactors and built-in command objects.

3. Debugging Tools

Debugging interactive applications requires additional mechanisms than supplied with conventional development environments. This has been a neglected area in other frameworks. Amulet provides an interactive *Inspector* that displays the object's properties, traces the execution of Interactors, pauses, single-steps and traces animations, and displays the dependencies of constraints and the properties of slots (see Figure 5). From the Inspector, programmers can also set breakpoints or have messages printed whenever the value of a slot changes. Furthermore, extensive error checking (when debugging is enabled) and helpful messages make Amulet applications easy to develop and debug. We try to make sure that programmers using Amulet never see "Segmentation fault" or other common but unhelpful C++ error messages.

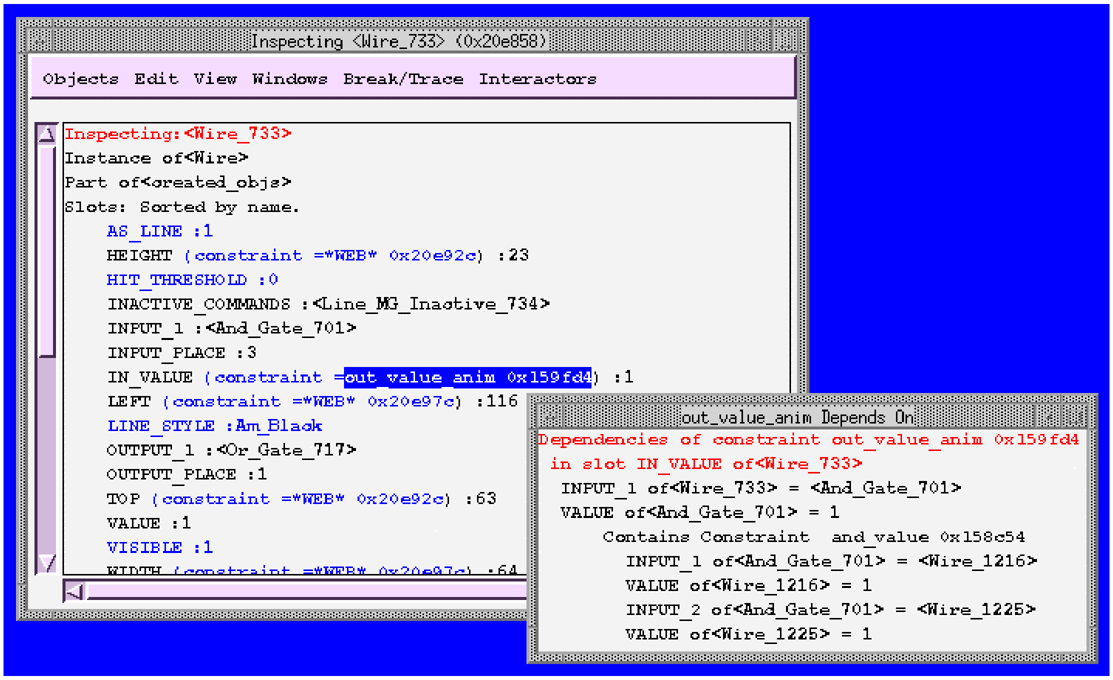


Figure 5: Inspecting a wire object, and the constraint in its `IN_VALUE` slot. The slots that are inherited are shown in blue and the local slots are shown in black. Notice that the high-level names of methods, constraints and objects are shown. Values can be edited by the programmer to see how the changes will affect the object.

4. Status and Future Work

The current version of Amulet (V3.0) has been released for UNIX, Windows NT, Windows 95, and the Macintosh (to get Amulet, see `http://www.cs.cmu.edu/~amulet`). Amulet has been downloaded over 11,000 times in the past year (over 200 times a week), and many research and commercial applications have been built using it.

In the future, we will be investigating techniques to support sound output, visualizations, World-Wide Web access and editing, and multiple people operating at the same time (also called Computer-Supported Cooperative Work -- CSCW). An important focus will be on *interactive tools* that allow most of the user interface to be specified without conventional programming.

5. Related Work

Amulet builds on many years of work on user interface toolkits and frameworks (see (Myers 1995) for a survey). It is primarily influenced by our previous Garnet framework (Myers 1990).

The first prototype-instance object system was probably ThingLab (Borning 1981), which did not use inheritance. Self (Ungar 1987) is the primary other language currently investigating the prototype-instance style. There are many differences between the Self and Amulet models. Self is its own language, so it does not have to integrate with an existing language. Self uses pure copy-down semantics, so after an instance is created, changes to the prototype are not reflected in the instances. Finally, Self does not support constraints. Other systems that have used the prototype-instance object model include Apple's NewtonScript and Sk8 languages, and General Magic's MagicCap. However, none of these support structural inheritance or constraints.

There are many research systems that support constraints. The first system with constraints was probably SketchPad (Sutherland 1963). Many systems have used constraints as part of an object system, but none is as general-purpose or fully-integrated as Amulet. The first integrated constraint and object system was ThingLab (Borning 1981), which supported multi-way constraints. Rendezvous (Hill 1994) was designed to help create multi-user applications in Lisp. Like Amulet, Rendezvous allows multiple one-way constraints to be attached to a variable, but Rendezvous requires that variables be explicitly declared and uses a different implementation algorithm. SubArctic (Hudson 1996) supports an efficient implementation of a few simple layout constraints in Java, but does not have a general-purpose constraint solver.

Using command objects to support undo was introduced in Apple's MacApp and has been used in many systems including InterViews (Linton 1989) and Gina (Berlage 1994). There is a long history of research into various new undo mechanisms, and Amulet is specifically designed to allow new mechanisms to be explored. The selective undo mechanism in Amulet is closest to the Gina mechanism (Berlage 1994), but adds the ability to repeat previous commands, and to undo selections and scrolling.

6. Summary

We are very excited about the effectiveness of Amulet as a useful platform with which to perform user interface research. We hope it will also continue to be popular for user interface education and for the implementation of real systems. The innovations in Amulet and the integration of novel object, constraint, input, output, command, and undo models, make it an attractive candidate for supporting both today's and tomorrow's user interfaces.

Acknowledgments

We want to thank the many users of Amulet who have helped us find bugs and improve the system. For help with this paper, we would like to thank Bernita Myers, Alan Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, and Patrick Doane.

References

(Berlage 1994) Thomas Berlage. “A Selective Undo Mechanism for Graphical User Interfaces Based on Command Objects,” *ACM Transactions on Computer Human Interaction. ACM Transactions on Computer Human Interaction.* 1994. **1**(3). pp. 269-294.

(Borning 1981) Alan Borning. “The Programming Language Aspects of Thinglab; a Constraint-Oriented Simulation Laboratory,” *ACM Transactions on Programming Languages and Systems. ACM Transactions on Programming Languages and Systems.* 1981. **3**(4). pp. 353-387.

(Hill 1994) Ralph D. Hill, Tom Brinck, Steven L. Rohall, John F. Patterson and Wayne Wilner. “The Rendezvous Architecture and Language for Constructing Multiuser Applications,” *ACM Transactions on Computer-Human Interaction.* 1994. **1**(2). pp. 81-125.

(Hudson 1996) Scott E. Hudson and Ian Smith. “Ultra-Lightweight Constraints,” *ACM SIGGRAPH Symposium on User Interface Software and Technology,* Proceedings UIST'96. Seattle, WA, Nov, 1996. pp. 147-155. http://www.cc.gatech.edu/gvu/ui/sub_arctic/.

(Krasner 1988) Glenn E. Krasner and Stephen T. Pope. “A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 system,” *Journal of Object Oriented Programming. Journal of Object Oriented Programming.* 1988. **1**(3). pp. 26-49.

(Lieberman 1986) Henry Lieberman. “Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems,” *Sigplan Notices.* 1986. **21**(11). pp. 214-223. ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'86.

(Linton 1989) Mark A. Linton, John M. Vlissides and Paul R. Calder. “Composing user interfaces with InterViews,” *IEEE Computer. IEEE Computer.* 1989. **22**(2). pp. 8-22.

(Myers 1995) Brad A. Myers. “User Interface Software Tools,” *ACM Transactions on Computer Human Interaction.* 1995. **2**(1). pp. 64-103.

(Myers 1998) Brad A. Myers. “Scripting Graphical Applications by Demonstration,” *Human Factors in Computing Systems,* Proceedings SIGCHI'98. Los Angeles, CA, Apr, 1998. pp. 534-541.

(Myers 1992) Brad A. Myers, Dario Giuse and Brad Vander Zanden. “Declarative Programming in a Prototype-Instance System: Object-Oriented Programming Without Writing Methods,” *Sigplan Notices.* 1992. **27**(10). pp. 184-200. ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'92.

(Myers 1990) Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish and Philippe Marchal. “Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces,” *IEEE Computer.* 1990. **23**(11). pp. 71-85.

(Myers 1996a) Brad A Myers and David Kosbie. “Reusable Hierarchical Command Objects,” *Proceedings CHI'96: Human Factors in Computing Systems,* Vancouver, BC, Canada, April 14-18, 1996a. pp. 260-267.

(Myers 1997) Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski and Patrick Doane. “The Amulet Environment: New Models for Effective User Interface Software Development,” *IEEE Transactions on Software Engineering.* 1997. **23**(6). pp. 347-365. June.

(Myers 1996b) Brad A. Myers, Robert C. Miller, Rich McDaniel and Alan Ferrency. “Easily Adding Animations to Interfaces Using Constraints,” *ACM SIGGRAPH Symposium on User Interface Software and Technology,* Proceedings UIST'96. Seattle, WA, Nov, 1996b. pp. 119-128. http://www.cs.cmu.edu/~amulet.

(Palay 1988) Andrew J. Palay, Wilfred J. Hansen, Michael Kazar, Mark Sherman, Maria Wadlow, Tom Neuendorffer, Zalman Stern, Miles Bader and Thom Peters. “The Andrew Toolkit - An Overview,” *Proceedings Winter Usenix Technical Conference,* Dallas, Tex, Feb, 1988. pp. 9-21.

(Sutherland 1963) Ivan E. Sutherland. “SketchPad: A Man-Machine Graphical Communication System,” *AFIPS Spring Joint Computer Conference,* 1963. pp. 329-346.

(Ungar 1987) David Ungar and Randall B. Smith. “Self: The Power of Simplicity,” *SIGPLAN Notices.* 1987. **22** pp. 241-247. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications; OOPSLA'87.

(Vander Zanden 1994) Brad Vander Zanden, Brad A. Myers, Dario Giuse and Pedro Szekely. “Integrating Pointer Variables into One-Way Constraint Models,” *ACM Transactions on Computer Human Interaction.* 1994. **1**(2). pp. 161-213.