

# Modular Implementation of Presentations

Pedro Szekely\*

Computer Science Department  
Carnegie-Mellon University  
Pittsburgh, PA 15213

## Abstract

The presentation of an application program specifies how the data and operations provided by an application are presented to users. Most traditional techniques for implementing presentations lead to unstructured, unmodular implementations that are hard to construct and change. We present a model of presentation that identifies the dependencies between the presentation and functionality portions of an application. Based on this model, we show how several implementation techniques can be used to construct presentations in a modular way.

## Keywords

Graphical User Interfaces, User Interface Management Systems, Semantics of Interaction, Object-Oriented Programming.

## Introduction

Separating the implementation of the user interface and the functionality of application programs into independent modules has many benefits. In addition to the standard benefits of modularity, separation facilitates experimentation with different user interface designs; allows the construction of multiple interfaces for a single program; supports the construction of the user interface out of reusable components that can be connected with the functionality portion of applications; allows the use of declarative and other techniques besides programming to specify the user interface.

In practice, the implementation of a program is sprinkled with calls to procedures to perform user interface tasks

\*This research was sponsored by the Defense Advanced Research Projects Agency (DOD), monitored by the Air Force Avionics Laboratory Under Contract AFOFR S-82-0219. 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 the Defense Advanced Research Projects Agency or the US Government.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

(e.g. display an error message, display a menu of commands, read an input event). As a result, interface decisions get scattered throughout the implementation of the program.

Consider the implementation of the output portion of a program. When a routine changes a data structure that is being displayed, all the information about the nature of the change is in that routine. Hence, it appears to be "easier" to implement the update of the display inside the routine, especially because it knows exactly what it changed, and can presumably update the image in the most efficient way. Unfortunately, this "ease" of implementation results in a complete lack of modularity.

This paper shows that the *presentation* (output) portion of an application can be implemented in a modular way. The paper has two parts. The first part discusses a model that explains what goes on in the presentation of an application, and identifies the unavoidable dependencies between presentation and functionality. The second part discusses several techniques to implement the communication between functionality and presentation. The techniques are compared with respect to the kind of modularity they provide, and with respect to the restrictions they impose on the kind of information that can be presented.

## The Structure of Interactive Programs

Interactive programs can be viewed as having the following structure. Their functionality is defined by a set of abstract data types that model the behavior of the objects in the user task domain. Instances of the types model the objects in the user domain, and the operations defined for the types model the actions users can perform on the objects.

Instances of data types, which we simply call *objects*, are hidden from the user. Objects are represented in a format suitable for implementing the operations, and must be converted to a format understood by the output devices before they can be presented to the user. Operations are not readily accessible to users either. Events generated by input devices must be converted into requests to invoke the appropriate operations. Thus, the objective of the user interface is to make the objects and operations conveniently accessible to the user.

## A Model of Presentation

We introduce our model of presentation in stages. We

start with a simple definition of presentation, and gradually embellish it to take into account the restrictions imposed by the characteristics of graphical, two-dimensional interfaces. For each definition of presentation we record the information required to construct a presentation mapping according to that definition. The information requirements for the final definition are used to identify the dependencies between functionality and presentation. For a different model of both input and output see Shaw [9].

### Initial Model

A presentation is a mapping that converts objects from a representation suitable for computation into a representation understood by the output devices. Typically, presentations are implemented on top of a graphics package, or on top of the graphics abstractions provided by a window manager. Thus, presentations need not be concerned with the idiosyncrasies of output devices. For the purpose of this paper, the precise details of the abstractions provided by specific graphics packages or window managers are irrelevant; we simply assume the existence of a type we call *Image* that represents these abstractions (we call *images* the instances of type *Image*).

A presentation mapping for a set of objects is specified by a set of rules that associate with each object an *image*. Since the rules defining a presentation need to extract from the objects the information to display, a presentation has to be specialized according to the type of objects they present.

Presentation mappings should be specialized to the abstraction defined by types, not to their implementation. Consider the type *Stack*. It can be implemented by an array, with the first element containing the index of the top of the stack, and the next lowest-indexed elements containing the entries in the stack. A presentation mapping for arrays *cannot* be used to appropriately display stacks because it wouldn't know to display only the active elements of the array, and it would also display the first element of the array, which is not part of the stack. Thus, it is necessary to give up the hope that appropriate presentations can be constructed automatically in terms of mappings associated with type constructors. Only a type-specific presentation mapping can present the instances of a type adequately.

The initial definition of presentation can be formalized as follows:

**PRESENTATION DEFINITION 1** *The presentation of objects of type T is defined by a mapping  $P_T$ :*

$$P_T : T \rightarrow \text{Image} \\ \text{object} \mapsto \text{image}^1$$

The subindex  $T$  in  $P_T$  indicates that presentation mappings are type-specific. The information needed to construct a presentation mapping is the following:

**INFORMATION REQUIREMENT 1** *Contents of the object presented.*

<sup>1</sup>The notation should be interpreted as follows:  $\rightarrow$  specifies the domain and range of the mapping in terms of sets;  $\mapsto$  specifies the domain and range in terms of elements.

### Multiple Presentations and Format

A crucial requirement of presentation mappings is the ability to define multiple presentations for objects of the same type. This requirement is illustrated by the pervasive use of multiple formats to display times and dates, files, graphs, and virtually any object defined by an application.

There are two ways to extend the definition of presentation mapping to support multiple presentations. The first is to allow formatting information to be passed to  $P_T$  via a parameter. The second is to define multiple presentation mappings for a single type. Small variations on the presentation are defined by the format parameter (e.g. font, background color), while completely different presentations are defined by different  $P_T$  (e.g. a mapping to display an integer as a string of digits and another one to display it as a bar-graph).

In theory, all possible presentations of a type could be specified by a single appropriately parameterized mapping. However, since radically different presentation styles are typically implemented by different procedures, a model including multiple presentation mappings captures reality better.

The following is the revised definition of presentation:

**PRESENTATION DEFINITION 2** *The presentation of objects of type T is defined by a set of mappings  $\{P_T^{s_1}, \dots, P_T^{s_n}\}$ , where  $s_1, \dots, s_n$  are different presentation styles defined for type T.*

$$P_T^s : T \times \text{Format}_{P_T^s} \rightarrow \text{Image} \\ (\text{object}, \text{format}) \mapsto \text{image}$$

Type  $\text{Format}_{P_T^s}$  represents the set of format values that can be used to control the appearance of the images produced by  $P_T^s$ . *Format* is qualified with the subscript  $P_T^s$  to indicate that format types are specific to individual presentation mappings.

Consider type *Number*. Numbers can be displayed in many different ways. They can, for instance, be displayed as strings of digits, or as dials. Since these two presentations are so different they are considered different styles, and thus are defined by different presentation mappings, called, let's say,  $P_{\text{Number}}^{\text{digits}}$  and  $P_{\text{Number}}^{\text{dial}}$ . Fine control of the formatting is specified by the format parameter.  $\text{Format}_{P_{\text{Number}}^{\text{digits}}}$  allows the specification of attributes like font, precision, French or American style decimal point, etc., while  $\text{Format}_{P_{\text{Number}}^{\text{dial}}}$  allows the specification of attributes like digital or analog, rounded or square, scale, etc.

The additional information required to construct a presentation mapping is the following:

**INFORMATION REQUIREMENT 2** *Style and format to control the presentation of the object.*

### Structured Types

Many objects used in application programs are structured: they are aggregates of other objects. Sets, queues, stacks, lists, hash tables, trees, graphs are just a sample of commonly used ones. In fact, by viewing an application as an instance of a single structured type, our definition of presentation mappings can be applied to complete applications.

The presentation of a structured object is specified by a mapping that calls the presentation mappings of its components. Suppose *IntStack* is a type implementing stacks of integers. A presentation for *IntStack* that displays the stack as a column of numbers can be defined by a mapping  $P_{IntStack}^{column}$  that calls one of the  $P_{Number}$  mappings (e.g.  $P_{Number}^{dial}$  or  $P_{Number}^{digits}$ ) for each element of the stack. Thus, the definition of  $P_{IntStack}^{column}$  has the form shown below; *S* and *f* represent the style and format for presenting the stack entries:

$$P_{IntStack}^{column}(aIntStack, format) = \dots \\ \dots \\ P_{Number}^s(\dots, f, \dots) \\ \dots$$

*S* and *f* should be variables, not constants. If *s* and *f* were constants,  $P_{IntStack}$  would only know how to display the stack entries in one specific style and format. If *s* and *f* are variables, they can be bound to values supplied by the caller of  $P_{IntStack}$  through the *format* parameter. The latter definition of  $P_{IntStack}$  is more general because it is parameterized with respect to the style and format for presenting the stack entries.

Thus, the following is an additional information requirement for the construction of presentation mappings:

**INFORMATION REQUIREMENT 3** *Format and style for nested presentations (refinement of requirement 2).*

Note that since nested presentation mappings may themselves call on other presentation mappings and so on, specifying the style and format values for the nested presentations can be very hard. Generic types complicate things more. Suppose *Stack* is a generic definition of stacks of any kind of objects.  $P_{Stack}$  should also be generic. It should only present the "stackness" attributes of a stack, and call another mapping to present the stack entries. Specifying the style and format values for the presentation of the entries is harder because their type is unknown.

The presentation mappings for structured types have an additional information requirement. Since nested mappings create images that are parts of a larger image, a mechanism is needed to combine the multiple images into a single image. For instance, if *Image* is a bitmap, then  $P_{IntStack}$  has to give  $P_{Number}$  the coordinates of the location within the global bitmap where it should draw the image. If *Image* is some kind of hierarchical data structure, then  $P_{IntStack}$  can take the image produced by  $P_{Number}$ , possibly scale it, and then integrate it with the rest of the image.

As the example with bitmaps illustrates, integrating the images produced by nested mappings can lead to an additional information requirement<sup>2</sup>:

**INFORMATION REQUIREMENT 4** *Portion of the complete image to create.*

### Updating Presentations

A fundamental characteristic of interactive interfaces is that when the objects being presented change, the display is updated to reflect the changes. The presentation of changes

<sup>2</sup>The current definition of presentation should have been modified to account for this requirement, but it was not. This requirement is a special case of state information, and the final definition takes it into account.

has two characteristics that are not captured by the current definition of presentation.

First, since images cannot be created instantaneously, invoking *P* to recreate the complete image of an object each time the object changes is unreasonable. The presentation mapping needs to know what aspect of an object changed, and only update the corresponding part of the image.

Second, updating the image of an object to correspond to its new state after a change might not be an adequate presentation of the change. For instance, suppose two elements of an array are swapped. Instead of presenting a snapshot with the elements in one position and then a snapshot with the elements in the new position, we might want to present an animation showing the elements being swapped [3]. The presentation of the change displays images which do not correspond to any state of the object.

A new definition of presentation is needed to capture the notion of *change* explicitly. We introduce a new mapping, called the update mapping *U*, whose purpose is to update images when objects change. The definition of presentation is now as follows:

**PRESENTATION DEFINITION 3** *The presentation of objects of type T is defined by two sets of mappings  $\{P_T^s\}$  and  $\{U_T^s\}$ .*

$$P_T^s : T \times Format_{P_T^s} \rightarrow Image \\ (object, format) \mapsto image \\ U_T^s : T \times Format_{P_T^s} \times Changes_{P_T^s} \rightarrow Image \\ (object, format, change) \mapsto image$$

The *P* mapping creates an image corresponding to the current state of the object, and it is called whenever the image should be recreated from scratch. The *U* mapping is called whenever the image should be updated to reflect a change. Type  $Changes_{P_T^s}$  is the set of changes to objects of type *T* that the presentation can display. Since different presentations of a single object can display completely different views of the object, the set of changes relevant to each presentation can be different. Thus, *Changes* is qualified with the  $P_T^s$  subscript to indicate that the *Changes* set can be specific to the type of object and the presentation mapping.

Let us return to the stack example. The set  $Changes_{P_{Stack}}$  could be defined to contain two elements, *push* and *pop*. The behavior of  $U_{Stack}(aStack, pop)$  would be to erase the top of the stack, and the behavior of  $U_{Stack}(aStack, push)$  would be to query *aStack* for the top entry, and call the presentation mapping for the stack entries to display the entry.

The following are other examples of the use of the *Changes* type and the *U* mapping:

$Changes_T = \{true\}$  The presentation only wants to know whether an object has changed. The *U* mapping can be defined to invoke the *P* mapping to regenerate the image from scratch.

$Changes_T = \{begin\_op_i, end\_op_i \mid op_i \in T\}$  The presentation cares about operations being invoked or ending. *U* can be defined to highlight an icon corresponding to the operation when it is invoked, and to un-highlight it when it ends.

$Changes_{Array} = \{swap(to, from), store, \dots\}$

When invoked with  $swap(to, from)$  as a parameter,  $U$  shows an animation of the corresponding elements of the array being swapped. When invoked with  $store$  the image of the corresponding element is updated.

INFORMATION REQUIREMENT 5 *Set of relevant changes.*

### Final Model

The implementation of the  $U$  mappings require that presentation mappings save state. For example,  $U_{Stack}$  needs to know the location of the images corresponding to the stack entries, their size, the format information used to create them, etc. Hence, instead of thinking about presentations as mappings it is better to think of them as objects. The state of presentation objects stores the information necessary to construct and update the connection between an object and an image. This is the information that we described as *information requirements*. The operations of presentation types are called to construct or update images. They correspond to the  $P$  and  $U$  mappings of the last definition.

**FINAL PRESENTATION DEFINITION** *The presentation of objects of type  $T$  is defined by a presentation type  $T_p$ .*

The state of a presentation type can be modelled as if stored in four variables:

*Object* A reference to the object presented.

*Image* A reference to the part of the image the presentation is responsible for.

*Format* The formatting information used to display the object. The formatting information is initialized when the presenter type is created, and is used by the  $U$  mapping to update the presentation.

*Local state* Presentation specific state. For instance, presentations for structured objects keep references to sub-presentations and a record of the way they are organized.

Presentation types provide two operations:

$P$  Recreates the contents of the image from scratch.

$U$  Takes as parameter a specification of a change, and updates the state and the image to reflect the change.

Also associated with the definition of  $T_p$  are the types  $Format_{T_p}$  and  $Changes_{T_p}$ , as defined in presentation definitions 2 and 3.

The presentation of an object is an instance of a presentation type. Presentations are created by instantiating and initializing a suitable presentation type. Multiple presentations of a single object are created by making multiple instances of, possibly different, presentation types.

A nice property of modelling presentations as types is that presentation instances are objects like any other object, and hence subject to be presented using presentation types. For instance, a presentation of a presentation  $p$  could display a menu with the format values to control  $p$ . Such a menu can be used to let the user dynamically select the format values.

The following summary of the information requirements of presentation types shows that only information requirements 1 and 5 refer to information that comes from the functionality. The other information comes either from the state of the presentation or is supplied as part of the definition of the presentation, and hence does not create dependencies between the functionality and presentation modules.

1. Contents of the object presented.
2. Style and format to control the presentation of the object.
3. Portion of the complete image to create.
4. Format and style for nested presentations (refinement of requirement 2).
5. Set of relevant changes.

### Implementation Techniques

In this part of the paper we discuss techniques for implementing the communication between the functionality and the presentations of an application. We compare these techniques with respect to the amount of modularity they provide, and with respect to the restrictions they impose on the kind of information that can be communicated.

#### Accessing Information to Construct Presentations

Techniques for accessing the information to be displayed can be divided into two classes:

*External* The presentation type is implemented as a stand alone type, separate from the type of the object being presented. Examples of systems that use external representations are: Smalltalk [4], Scofield's UIMS [8] and MacApp [2].

*Internal* The presentation type is part of the type of the object being presented. Examples of systems that use internal representations are: "traditional" programming techniques, Wallis [14] and Descartes [10].

The difference between the two classes is that internal implementations are part of the implementation of the type of the object being presented, and hence have direct access to all the information about the object. External implementations have to use the operations provided by the type to access the information.

#### Generality

Internal implementations are more general than external ones because types may not provide enough operations to access the information needed for display; they may only provide the operations that define the abstraction embodied by the type. For instance, typical definitions of type *Stack* provide operations *push*, *pop* and *is\_empty*, which are not appropriate to access all the entries in a stack. An iterator is needed to access all the stack entries without popping them.

We don't view the requirement of extending types with operations to access information needed for display as a great disadvantage of external implementations. Such operations provide a clean interface for interacting with the information in the type instances.

Runtime overhead is not a good argument against external representations either. All clients of a type pay the

same overhead when interacting with objects via the type operations. If the overhead is too large for display purposes then it is probably too large for the other uses of the object, and the given implementation of the type would be useless anyway.

### Modularity

From the point of view of modularity external implementations are better than internal ones. All the usual benefits of information hiding are valid here. The following are properties of external implementations that internal ones don't have (or have to a lesser degree):

1. The implementations of the functionality types and the presentation types are hidden from each other. The dependency of the presentation on the functionality is explicit, since it is defined by the set of operations used to access the relevant information.
2. It is possible to experiment with different presentation types without having to change the code that implements the functionality. This is the most important advantage from the point of view of easing the implementation of user interfaces.
3. A single presentation type can be used with different implementations of a functionality type.
4. Different implementation languages can be used for the implementations of the functionality and the presentation types. For instance, the functionality can be implemented in Pascal or C, and the user interface, or part of it, can be specified with an interactive language like that of the Macintosh resource editor [1].
5. External implementations provide better support for sharing code among presentation types. The presentation types can be organized in a hierarchy [8] without forcing the functionality types to be in the same hierarchy<sup>3</sup>.
6. The state of the presentation is in a different object from the state of the functionality, and hence is more amenable to migration across machines. Thus, external implementations provide better support for architectures in which the user interface and the functionality of an application run on different workstations.
7. Presentation types can be implemented in isolation, i.e., without being associated with any other type (e.g. a presentation type defining a map of the US). Such isolated presentation types can be customized to access the information to display from a large number of unrelated types (e.g. the information to display about each state could come from any type that provides operations to access information about states).
8. Presentation types can have identical programming interfaces (i.e. operations with the same name). Thus, different presentation mappings can be attached to an object without the object knowing which mapping it is talking to. The different mappings can be tailored to different classes of users and to different output devices (even hardcopy and files).

<sup>3</sup>The advantage is not as strong if multiple inheritance is provided.

### Interesting Changes

In this section we discuss techniques for tying the *U* mappings with the functionality. The goal is to call the *U* mappings with the appropriate parameters when the relevant changes occur.

Once again, we distinguish between two classes of implementation techniques:

*Announcement* The functionality calls routines to announce changes of interest to the presentation. We distinguish between two classes of announcements:

*Indirect* The announcement is an element of the *Changes* type. A registry mechanism is used to bind the announcements to the *U* mappings. Examples of systems that use indirect announcements are: Balsa [3] which uses what its authors call *interesting events*, and Smalltalk [4] which uses what is called the *changed:* message.

*Direct* Announcements are made by directly calling the *U* mapping with the relevant parameters. The built-in output facilities of most traditional programming languages use direct announcements: `printf` in C [5], `format` in Lisp [11], etc.

*Recognition* Changes are recognized by the presentation types themselves. The idea is that the presentation types can *watch* the behavior of the functionality and recognize when a change of interest to them has occurred. Some techniques that can be used to recognize changes are:

*Active values* When a value is made active any attempt to access or store it triggers the invocation of a routine (e.g. Descartes [10] and Loops [12]). The presentation recognizes changes by making the relevant objects active values<sup>4</sup>.

*Daemons* Daemons are defined by wrapping code around the type operations so that each time the operation is executed other routines are called, before and/or after (e.g. Flavors [6]). The presentation recognizes changes by installing daemons on the operations that make the relevant changes.

A technique similar to daemons is used in MacApp. Operations are invoked from *command* objects, which are part of the user interface. The command object announces the changes done by the corresponding operation by calling a routine specifying the portions of the display that need to be updated<sup>5</sup> [2].

### Generality

Announcement is more general than recognition. Any change can be announced by inserting a call to a routine that makes the announcement in the right place in the functionality, and supplying it with the appropriate parameters.

<sup>4</sup>The implementation of active values is technically an indirect announcement. However, since they are implemented at a very low level, programs don't need to invoke the announcements explicitly.

<sup>5</sup>Of course, these "daemons" won't be invoked if the operation is not called from a command object. Nevertheless, the method works well for the implementation of interactive applications.

Recognition techniques based on active values have the following shortcomings:

- Active values can only recognize changes in the state of objects. However, in many cases the complete state of the routines is not represented as objects, and hence active values cannot recognize changes in it. For instance, active values cannot typically recognize invocation and termination of operations.
- With active values it is difficult to recognize changes at the right level of abstraction. The problem is similar to that of trying to specify the presentation of a type based on the presentation of the type constructors: active values recognize changes in the data structures that implement the abstractions; they do *not* recognize changes from the point of view of the abstraction embodied by the type. Consider again the implementation of *Stack* based on arrays. Updating the array cell containing the index of the top of the stack, and the cells containing the stack entries should *not* be considered separate changes. From the point of view of the stack abstraction they are both part of a single change.

Recognition techniques based on daemons have the following shortcomings:

- Changes that occur during the execution of an operation cannot be recognized. Operations are viewed as atomic (except if they call another operation). For instance, daemons do not provide adequate support for the implementation of progress reports [7].
- Detailed descriptions of changes cannot be recognized. With daemons it is possible to recognize that an operation was invoked and that it terminated, but it might not be possible to recognize what it did. For simple operations, the parameters with which an operation is called may be enough information about the change it will perform, but that is not always the case.

### Modularity

Announcements are not modular because their implementation require changing the functionality implementation. Announcements have the following disadvantages with respect to recognition:

1. It is necessary to find the appropriate places in the functionality implementation from where the announcements should be made. This can lead to the common error of some routine changing a displayed object, but forgetting to call the relevant update routine.
2. If the functionality code is sprinkled with calls to procedures to make announcements the same code cannot be used in contexts where the announcements are not necessary, or in contexts where different kinds of announcements are necessary.

However, indirect announcements are more modular than direct. At least the identity of the *U* mapping is not wired down into the implementation of the functionality.

### Final Remarks

According to our model and our analysis of implementation techniques, presentations should be implemented using external implementations of the presentation types, and if possible changes should be implemented based on recognition. There are cases when indirect announcements are necessary (e.g. Balsa program animations), but direct announcements and internal implementations should be avoided.

Finally, a more detailed description of the model presented here, a model of input and a description of a prototype UIMS based on these models can be found in [13].

### Acknowledgements

I thank Phil Hayes and Ed Smith for useful discussions on the issues reported in this paper, and Richard Cohn, David Garlan, Dario Giuse and Bruce Horn for helpful comments on earlier drafts.

### References

- [1] Apple Computer. *Inside Macintosh*. Addison-Wesley, Reading, Mass., 1985.
- [2] Apple Computer. MacApp. (MacApp Examples).
- [3] M. H. Brown and R. Sedgewick. A system for algorithm animation. *Computer Graphics*, 18(3):177-186, July 1984.
- [4] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Mass., 1983.
- [5] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.
- [6] D. Moon and D. Weinreb. *Lisp Machine Manual*.
- [7] B. Myers. The importance of percent-done progress indicators for computer-human interfaces. In *CHI'85 Conference Proceedings*, 1985.
- [8] J. A. Scofield. *Editing as a Paradigm for User Interaction*. PhD thesis, University of Washington, August 1985.
- [9] M. Shaw. An input-output model for interactive systems. In *CHI'86 Conference Proceedings*, 1986.
- [10] M. Shaw, E. Borison, M. Horowitz, T. Lane, D. Nichols, and R. Pausch. Descartes: a programming language approach to interactive display interfaces. In *Proceedings of the Symposium on Programming Language issues in Software Systems*, June 1983.
- [11] G. L. Steele Jr. *Common Lisp The Language*. Digital Press, 1984.
- [12] M. J. Stefik, D. G. Bobrow, and K. M. Kahn. Integrating access-oriented programming into a multiparadigm environment. *IEEE Software*, 3(1), January 1986.
- [13] P. A. Szekely. Separating the user interface from the functionality of application programs. PhD Thesis (In preparation).
- [14] P. Wallis. External representations of objects of user-defined type. *ACM Trans. Prog. Lang. Syst.*, 2(2):137-152, April 1980.