View binding and user enhanceable systems[†]

Harold Thimbleby
Stirling University
Scotland, FK9 4LA.
Email: `hwt@uk.ac.stir.cs`

Thursday, October 1, 1998

ABSTRACT

Most user interfaces are constructed by programs, so there is no direct relationship between the program, its structure and the user interface it implements. For example, graphics drawn on a screen need have no correspondence with regions of the screen sensitive to the user's input. *View binding* is introduced as a binding scheme (related to static or lexical binding schemes) whereby the user's browsing the user interface also manipulates the program's environment in a particular way. The result is a direct relation between user interface and program, with many advantages.

Like particular binding schemes in programming languages, view binding is not a solution to every programmer's or user's wishes, yet it provides a useful discipline that can help simplify user interfaces and their programs. In particular, it is very suitable for user enhanceable systems.

No system currently available is implemented with a strict view binding scheme, but this paper exhibits suggestive examples taken from several well-known programming languages and interactive systems.

Keywords: view binding, view scope,
constructed interface, user enhancement

[†] To appear in *The Visual Computer: An International Journal of Computer Graphics*, Springer Verlag.

“Research into articulate graphical communication has just begun to scratch the surface, but the paradigm of operations possessing both aesthetic and mechanical semantics is a powerful approach.” D. C. Smith [17]

1. INTRODUCTION

User interfaces are the consequences of running interactive programs. Unfortunately, in general, there is no essential correspondence between the program text and the user interface the running program generates.

This paper introduces a notion, *view binding*, where the graphical structure of the user interface and the program’s environment (identifier bindings) do correspond closely. This approach obtains advantages for flexible and reliable interactive systems programming, particularly for safely modifying existing designs, for example to accommodate changing user requirements. This paper also discusses *user enhanceable systems* as such, and the relevance of view binding to simple and safe user enhancement.

In almost all interactive systems, the user interface is constructed by the program, by calculations that determine the size and location of rectangles, windows and other graphical objects. There is no significance to the program itself in these calculations; and there is little significance more generally except in the aesthetic impact on the user. A human factors interface evaluator may suggest improvements to an interface depending on increasing or decreasing certain values, intending to resize, recolour or reposition objects. Often this will have a ‘knock on’ effect, as when a mouse target region is moved: the *different* part of the program detecting mouse clicks in that region will also have to be modified correspondingly. In other words, the graphical regions of a system and the active regions of a system are computed independently. Of course, many languages provide abstractions where the graphical region and the active region are simultaneously controlled by the same parameters, but there is no necessary relation imposed by the programming language itself. A review of object oriented graphics programming can be found in [24].

In systems providing direct manipulation for modifying the user interface (HyperCard being a well-known and proprietary example of this approach, which will be discussed below), a user may be able to drag a composite graphical/active region as a tied unit. In such systems it would be unusual for the (high level) program to contain geometric calculations, since it would be difficult to maintain them as objects are dragged.

In discussing the design of Boxer, diSessa [7] suggests that the user ought to be able to act as a ‘naïve realist’ — to be able to take what is viewed on the screen as the system itself rather than as an interface to some underlying (and therefore arbitrarily complex and structurally unrelated) system. The intention is to obtain a strong WYSIWYG (what you see is what you get) principle [23], hence to reify

> “[…] a pervasive spatial metaphor in which language structures and relations are expressed by the spatial relations one sees on the screen. The intent is to tap the well-developed pool of knowledge about space that humans already possess in order to facilitate their making a model of the computational environment.”

Thimbleby [23] introduces the term *reflexive interaction paradigm*, where the programming language paradigms carry over into the user interface. An example of a poor reflexive interaction paradigm is fixed length arrays in Pascal being reflected in the user interface by (to the user) arbitrary restrictions on the number of characters they can type. A good example of a reflexive interaction paradigm is a *hyperprogramming* language [4] where the components of the program *are* components of the user interface, such as windows, menus and buttons. Because the program components in a hyperprogramming language are part of the user interface, as the user does things, such as selecting a different graphical object, the associated program components become deactivated and activated automatically. So as the user browses, the objects available, both to programs and to the user, change in concert. Thus, as the user goes from one window to another, the system 'keeps track' of which buttons it should display and which of the associated parts of program should be available. If the system keeps track in a particular way (to be defined) we will call the approach *view binding.* (Reference [4] used the slightly more awkward term *browse binding.*)

View binding will be seen to be a particular approach to tying the graphical interface with the program. View binding is an aspect of a programming system's paradigm, so it does not appear explicitly in a program: it is available to the programmer efficiently, reliably and (as it has no notational counterpart) it cannot be easily or accidentally circumvented.

2. BINDING AND SCOPE IN EXISTING SYSTEMS

Binding is the association of a name with a meaning or value. Binding is a fundamental concept in programming. Different systems, however, take different approaches to binding. The usual aim of binding schemes is to reduce the complexity of programming, to improve the ability to detect or prevent errors mechanically, to satisfy psychological needs of naming and organisation. These are aims we would also want to support in user enhanceable systems, but for the user. Dijkstra [6, Chapter 10] provides background and motivation.

In *static binding*, or lexical binding, the meaning of a specific name can be ascertained statically, that is without running the program. In a simple case, as in the programming language C, this may mean that by reading a program backwards and upwards from the occurrence of the name in which one is interested, the first declaration of that name found gives the appropriate meaning of that name. There may be other meanings for the same name, but these in turn will occur earlier in the program than the region we have been considering. The region of a program where a name has the particular meaning introduced by a declaration is the *scope* of that declaration (for that name).

In many languages, such as Pascal, scopes are constructed in syntactic *blocks*, typically surrounded by matched pairs of **begin** and **end** symbols, and which may be nested. Ada permits names to be decorated with the name of the relevant declaration; HyperCard permits expressions (not just names) to be evaluated in named scopes. We will discuss Pascal and HyperCard more fully below.

In *dynamic binding* the meaning of a name cannot in general be ascertained from a static analysis of the program, though for simple programs there may be little difficulty in doing so. The main difference between dynamic and static binding is that in dynamic binding one reads backwards down the run time procedure call stack rather than (in static binding) reading up the program text. In dynamic binding, the meaning of a free name in a procedure is not fixed, as the procedure can be called from other procedures each applying different bindings for that name. In contrast, in static binding each applied use of a name has a unique associated binding, the corresponding declaration of which can be uniquely determined by examining the (static) program text, rather than its run time behaviour.

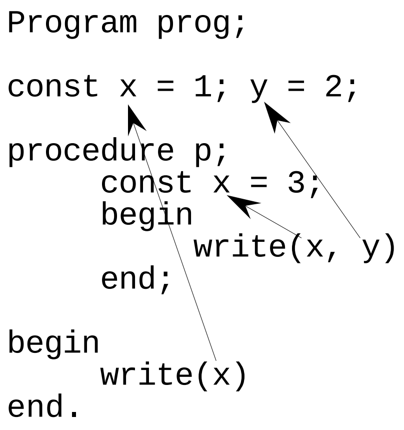
In short: in static binding, the meaning of a name in a procedure is determined by the bindings in effect where that procedure itself was *defined.* Whereas, in dynamic binding, the meaning of a name in a procedure is determined by the bindings in effect where that procedure itself was *called.* A procedure can only be defined in one place (in almost all languages) but it may of course be called in many places. This is one reason for debate about the relative merit of dynamic and static binding: static bindings are more tractable, but conversely procedure definitions are not so flexible. Furthermore, anything that can be done with dynamic binding can be simulated in a static binding regime by passing more parameters (though one has to know *before hand* what those additional parameters should be). Dynamic binding, then, is ideal for extensible and extending programs that are developed in an *ad hoc* or unpredictable (e.g., user or need driven) fashion.

Advances in program inference have reduced the need for dynamic binding at the expense of complicating static binding rules; on the other hand, the intention is to make analysis of the program tractable and reliable, whether performed by a human (as in conventional static binding) or by an automated language processor (as with type inference).

An important point is that a binding scheme is defined by the programming language's *paradigm*: it is not and should not be simulated [22]. Because it is not simulated, because it is not an optional 'discipline' or programming method used at the programmer's discretion, it will be *correctly* and *efficiently* implemented — by the system rather than the programmer (or user).

2.1. From Pascal binding to view binding

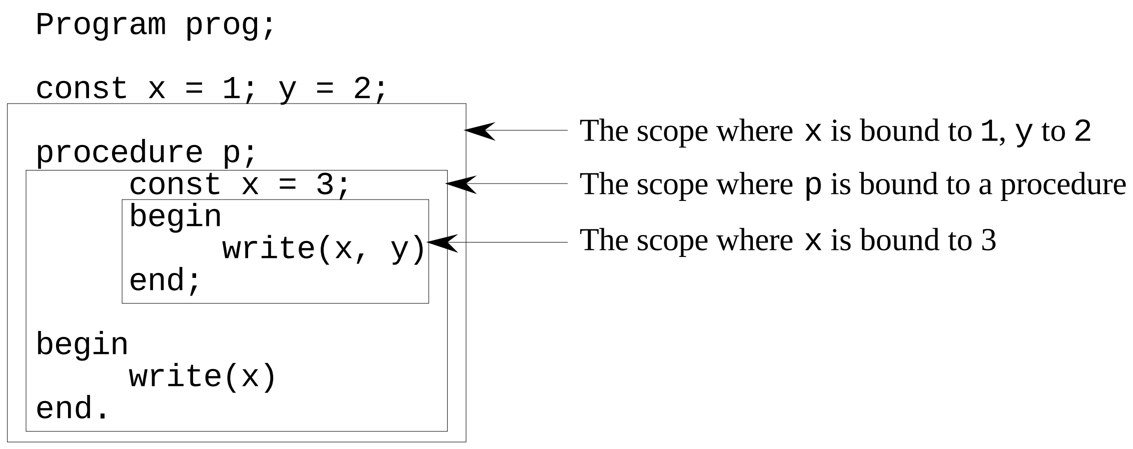
Binding arrows [20] can be used to indicate the binding occurrences of names from where they are applied. In figure 1, a simple Pascal program has been decorated with such arrows, making clear which bindings `x` and `y` are associated with in the two separate calls to the procedure `write`.



```
Program prog;

const x = 1; y = 2;

procedure p;
     const x = 3;
     begin
          write(x, y)
     end;

begin
     write(x)
end.
```

***Figure 1*. Binding arrows in Pascal.**

A Pascal program consists of invisible scopes, wherein each occurrence of a name has the same binding. We can use rectangles to indicate scopes explicitly (figure 2), with the proviso that smaller rectangles contained within larger rectangles impose their scope: otherwise known as *hiding* (or *shadowing* in LISP).



```
Program prog;

const x = 1; y = 2;

procedure p;
     const x = 3;
     begin
          write(x, y)
     end;

begin
     write(x)
end.
```

The scope where `x` is bound to `1`, `y` to `2`
The scope where `p` is bound to a procedure
The scope where `x` is bound to 3

***Figure 2*. Lexical scopes in Pascal.**

The precise idea of view binding is now easy to express.

In programming languages the scopes are conceptual and invisible but the program is explicit; in user interfaces supporting view binding, the scopes are explicit (i.e., real graphical objects, such as rectangles) but their associated program is invisible.

Rectangles are merely suggestive: why not consider them polygonal regions, windows, icons, or cells in a spreadsheet? As with conventional program scopes, the regions may be nested (e.g., windows containing dialog components) and tessellated (e.g., a spreadsheet). Thus the generality of view binding is considerable.

The lexical scopes of a language, like Pascal where they can be represented by imaginary rectangles, can be realised as actual *views* in a user interface. If this is done, a *view scope* is the view associated with a particular program scope. Thus as the user or program changes the view of a running program, the bindings of names may change: because views change, view scopes change, hence conventional scopes change. Conversely, as conventional scopes change (e.g., with procedure invocations), the graphic views change, maintaining the view scope correspondence. Of course, if one was to do this in Pascal or some other conventional language, any correspondence of static binding scopes and view scopes would be merely simulated, and not part of the paradigm of the language.

Lexical scoping permits a programmer to overload identifier names for different purposes in different parts of a program — the `x` in the Pascal program (figure 1 or 2) is bound in one scope to one value and in another scope to another value. In view scoping, it becomes convenient to use the term *event* for bound names. The same event such as 'mouse up' may be bound in one view to perform one activity, and bound in another view to perform another. In conventional binding schemes, events are so overloaded that the normal level of description is the function (e.g., what the mouse up event does in various places in the system) that is activated by the event.

2.2. Disciplines defined, and disciplines that encourage view binding

A *discipline* is a discretionary approach to programming: a programming language, although perhaps not enforcing a particular programming approach, may be used to simulate the approach. Thus LISP can be used to support a purely applicative style of programming if one is so inclined to do so, disciplining oneself to avoid all imperative features. Indeed, LISP gives no help to a programmer choosing to be so disciplined. In a programming language such as Pascal, any self-imposed discipline to program applicatively would be overwhelming, since imperative features are almost impossible to avoid, indeed the language does not *directly* support many basic applicative concepts (such as first class functions). Likewise, in all current interactive language systems, view binding is a discipline, but the degree of discipline required of a programmer varies considerably. Typically, less discipline is required in object oriented languages.

Abstract data views (ADV) [5] is a programming discipline that encourages connection between a program's object orientation and its user interface. It may be contrasted with view binding: it is a formal approach that is not necessarily implemented in the programming paradigm. It is a way of reifying abstract data types (ADT) in the user interface. Of course, some programming languages provide for ADTs within their notation (with varying degrees of thoroughness), and languages may be developed to provide ADVs also.

Another approach is abstraction-link-view (ALV) [8], which uses a constraint approach to link abstractions to views. The intention in ALV is to *separate* the application from the interface (a useful goal for user interface management systems and for multi-user applications); hence the structure of the user interface has no bearing on the program's.

3. EXAMPLES AND APPROXIMATIONS TO VIEW BINDING

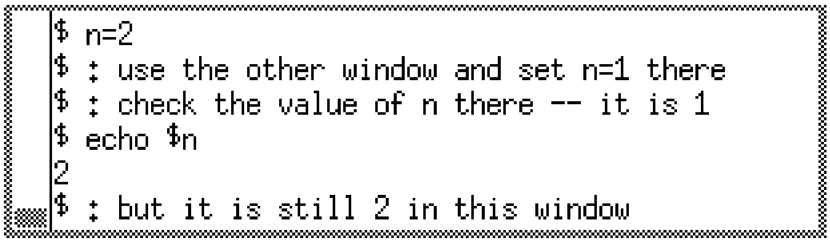
Of the many systems that *almost* support view binding, perhaps the Andrew Toolkit (ATK) is the most accessible [15]. In ATK, graphical objects (including text) can be embedded and they are subject to 'parental control,' which amounts to inheritance from the component that contains the embedded object.

Rather than describe ATK, Boxer [7, *op cit*] or the many other comparable systems in detail, and how their integrated approaches differ in detail from view binding, it is more straight forward to exemplify view binding by a sequence of simpler (and possibly more familiar) examples of increasing complexity. This sequence of examples is followed, in §3.4, by a brief examination of some alternative approaches.

3.1. Unix in a multi-window environment

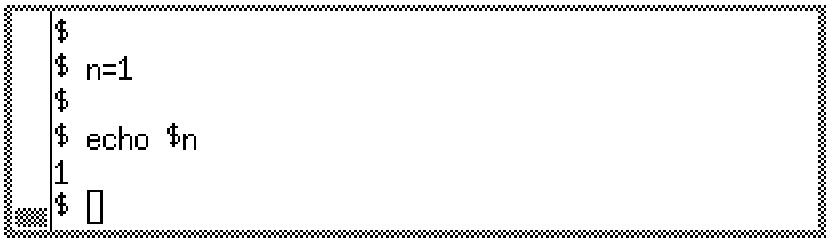
The Unix shell is one of many command-based systems that may be run under window systems, such as X. In these windowing systems, each window represents a separate shell (such as the Bourne Shell [2]), so creating a variable `n` in one window has no effect on its bindings in another window. In other words, the binding arrows in Unix windows are always constrained to start and end *within* windows.

Figure 3 shows a user declaring a variable `n` in each of two windows; in figure 3a, `n` is assigned the value `2`, *then* in figure 3b the same name `n` is bound to a different variable and assigned the value `1`. On returning to the first window (figure 3a), the value of `n` is still `2`, indicating that it is bound to a different variable. Of course, the interleaving of such actions between windows is difficult to narrate in a static medium such as this paper; but we will give several other examples.



```
$ n=2
$ : use the other window and set n=1 there
$ : check the value of n there -- it is 1
$ echo $n
2
$ : but it is still 2 in this window
```

***Figure 3a*. A variable declared and initialised to 2. The variable is unaffected by changing views to figure 3b and giving the same name a different value.**



```
$
$ n=1
$
$ echo $n
1
$ ▯
```

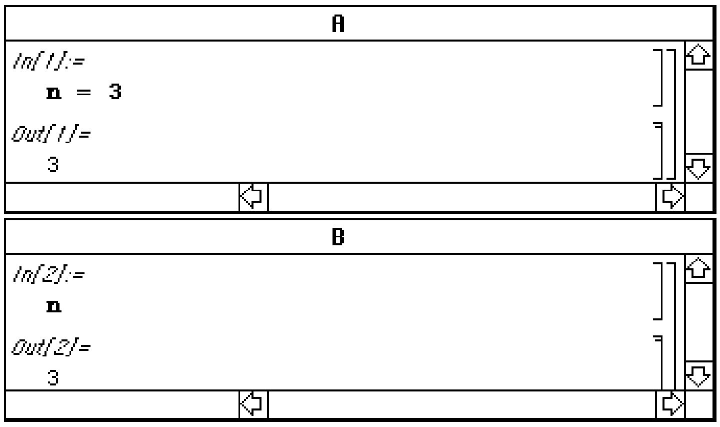
***Figure 3b*. Intermediate steps declaring and initialising a different variable with the same name in a different window. The actions in this window occur in a different view scope from those in the window shown in figure 3a.**

3.2. Mathematica

Mathematica [25], as typical of many window-based programming systems,* permits several windows to be open at once. Unlike a windowed Unix, each window in Mathematica is a window onto the *same* Mathematica environment. Thus a binding in one window (A) affects bindings in another (B); see figure 4. In other words, binding arrows *cross* window borders.

* The relevance of Mathematica for our discussion is its user interface behaviour, which is common across many platforms. Mathematica, as a language, is based on rewrite rules and has an obscure approach to binding (it is not a conventional programming language) but this issue is not related to the user's environment of windows, nor is it relevant to our present example.

Fortunately Mathematica numbers input expressions, so despite the user being permitted to interleave bindings in any order across windows, the scope of a name can be determined by going in reverse order over numbered expressions.



```
In[1]:=
   n = 3
Out[1]=
   3
```

```
In[2]:=
   n
Out[2]=
   3
```

***Figure 4*. Two Mathematica windows. `n` is bound in window A and subsequently applied in B.**

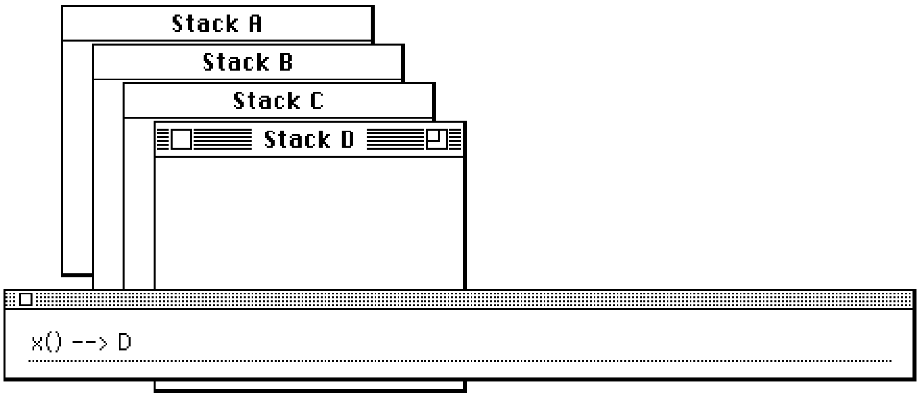
3.3. HyperCard

Like Mathematica, HyperCard [4] permits the user to have several windows (so-called *stacks*) open. Since we want to emphasise the integration of views and scopes in this paper, we do not need to distinguish the textual programming language (HyperTalk) from the entire, and mostly graphical, system (HyperCard). The reader is referred to [4] for definitions of terms such as *card* and *background*, which will be used in the discussion below.

In static binding, name bindings are searched by following the lexical block structure; in dynamic binding, the run time procedure nesting is followed. In HyperCard, which almost uses view binding, the visual nesting of objects is used; HyperCard uses the term *message path* for the route for searching (a more conventional term is *access chain* [1]).

Figure 5 shows four stacks, called Stack A to Stack D. The current stack (the stack the user most recently clicked in) is selected and highlighted, indicated by horizontal lines in its top bar. Each of the four stacks has defined the function `x()` to return a different value. For convenience, this value is the last letter of the stack's name.

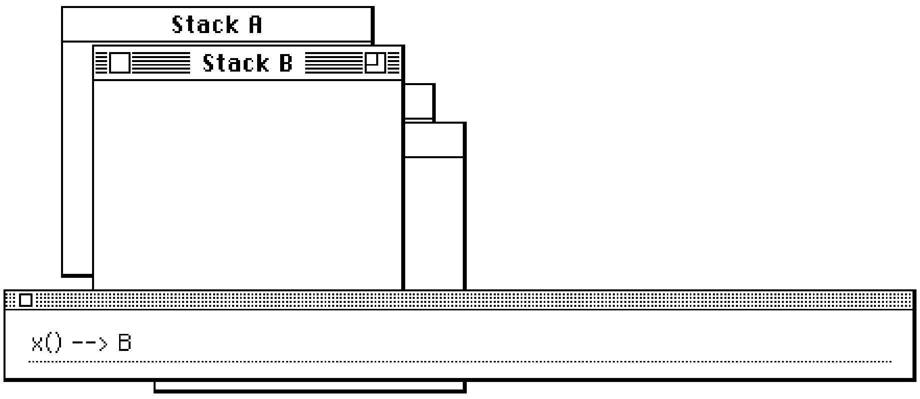
The figure shows that if the user submits the expression `x()` in the message window (the wide window) when Stack D is selected, the value of `x()` is `D`. In other words, when Stack D is selected and `x()` is bound in Stack D, that binding in that stack obtains.



```
x() --> D
```

***Figure 5*. Stack D is selected.**

Figure 6 shows the outcome when a different stack is selected, in this case Stack B.



```
x() --> B
```

***Figure 6*. Stack B is selected.**

The message window (into which the user types to evaluate expressions) is not geometrically within the currently selected stack. This is a convention of HyperCard which is indicated by the distinctive border of the window: clicking on such a window does not change the current view.

Figure 7 shows another HyperCard stack (called Browsing) currently showing one card, the third of four. There are buttons called “Go previous” and “Go next.” Clicking on these buttons takes the user to the previous or next card. The two buttons are so-called *background buttons*, meaning that they are bound in the background and appear on every card in that background.

The implementation of the button “Go previous” *might have been*:

```
on mouseUp
  go previous
end mouseUp
```

which means that when the `mouseUp` event occurs with the mouse positioned over this button, execute a `go previous` command, which changes the current card to the one previous to it.

When the user is on the first card, however, we do not want this behaviour. One solution is to implement the button to take account of this condition explicitly, something like:

```
on mouseUp
  if number of this card > 1 then go previous
  else beep
end mouseUp
```

Another solution is to place a button on top of the “Go previous” button in the first card alone. This button then intercepts the `mouseUp` event and can implement it differently from the background button. But HyperCard allows an elegant solution taking advantage of view binding. The button can invoke a procedure, `goPrevious`, implemented in the *background* scope to change cards. The first *card* scope can provide an alternative definition of `goPrevious`, to beep.

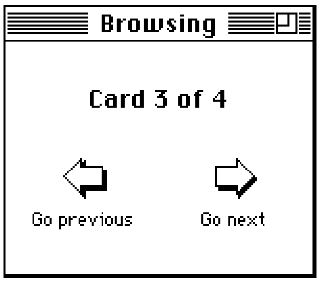


***Figure 7.* A simple HyperCard stack.**

The three procedures are given in full below with explanations of their effect on the environment.

```
on mouseUp -- background button binding mouseUp in view scope of the button
  goPrevious
end mouseUp
```

Within the view scope of the button “Go previous,” the user-originated event `mouseUp` is bound to a procedure with body `goPrevious`. As this is a background button, this binding occurs on all cards that are members of that background, because these cards have views contained in the wider view of the containing background.

```
on goPrevious -- background binding of goPrevious
  go previous
end goPrevious
```

Within the view scope of the background, the program-originated event `goPrevious` is bound to a procedure with body `go previous`. As this is a background binding, it is in view scope for all cards and buttons members of that background.

However, the first card hides this binding in its own view scope:

```
on goPrevious -- first card shadowing of goPrevious
  beep
end goPrevious
```

Within the view scope of the first card, `goPrevious` is bound to a procedure with body `beep`. As this is a card binding, this binding is in view scope for all buttons (and fields) members of that card’s view. It is not in view scope on any other card.

Changing the stack, by inserting new cards or deleting cards, can affect the binding of `goPrevious`, possibly with undesirable effects. Deleting the first card removes the card’s bindings, and therefore introduces an error in the “Go previous” button of the previously second card. Of course, this criticism is really a criticism that deleting cards etc has unobservable side effects. A solution would be to specify views by expressions that are maintained as invariants; a programmer easily can achieve this discipline by restrictingavailable operations.

The relationship between direct manipulation and invariants is addressed elsewhere in the literature on programming by demonstration [14].

3.4. Contrasts to view binding, and other approaches

Visual Basic [13] is superficially like HyperCard, differing mainly in its terminology, lack of persistence, and its use of Basic instead of HyperCard’s language, HyperTalk. However, Visual Basic does not have a seriously structured environment. (Visual Basic also has archaic constructs such as `COMMON`, which can be used to circumvent typing.) The name space for graphical objects, too, is flat. Whereas HyperCard can have many `mouseUp` handlers (each bound differently in different view scopes), Visual Basic constructs distinct names, such as `object1_click` or `object2_click`. Although there is a system event, click, that is automatically directed to appropriate objects, there is no *program* concept (identifier or value) ‘`click`’ that the system can bind in different scopes. Although Visual Basic has hierarchical graphics (parent controls and control arrays), it cannot modify or generate view scopes at run time, as can HyperCard, which does so easily (for example) by generating new cards. HyperCard can cut a card from one parent (i.e., background or stack) and paste it into another, thus changing its containing environment.

Unlike HyperCard, Visual Basic has no support for run time programming and therefore would not gain as much from a view binding regime. The programmer of Visual Basic has a shallow, fixed environment, which can be planned statically; in HyperCard, the environment can change at run time and, moreover, may be extended fairly safely at run time by users with comparatively limited knowledge of the overall program structure. (We can only say ‘*fairly* safely’ because, of course, HyperCard is not strictly view binding.)

Smalltalk is the pre-eminent object oriented language; although it is object oriented, user interfaces constructed with it have no essential relation with the program. Indeed many have argued that graphical representation and the programmer’s inheritance should be separate [e.g., 9]. One consequence is that the Smalltalk browser is an independent subsystem, structurally unrelated to the programs it browses.

Many graphical browsers for Smalltalk have been suggested [e.g., 10]. These browsers draw a graph representing the class structure, and in some cases the dynamic behaviour of the program. In all cases, what is drawn is a conceptual map of the program and, at best, user actions directed at nodes in the graph enable the user to edit corresponding components of the program. In view binding, we would expect user actions to be implemented by *those* components or others on their inheritance path.

The term ‘object oriented’ is applied to programming languages and to graphics in different ways, and as such the terms are unrelated. An object oriented programming system may or may not support object oriented graphics, and vice versa. Thus, for example: Smalltalk uses bit-mapped graphics (a paint model); HyperCard uses both paint and graphical objects; Pogo [19] uses object oriented graphics alone.

Although Pogo has a declarative approach for constructing user interfaces, it separates the programming language’s objects and inheritance structure from the graphical objects and view structure: view binding is available as an optional discipline, but is not guarantied by the paradigm (as it is in HyperCard). As with static binding in programming languages, if a way to circumvent view binding is provided the result is not view binding. View binding is distinguished by what it does *not* permit; though the things it does *not* permit may be more general or more convenient for certain applications.

Pygmalion was an innovative and ground-breaking interactive program-by-example system implemented in Smalltalk 72 [17]. Pygmalion could be said to be icon-oriented, and was programmed by direct manipulation actions on icons. Like our notion of view, Pygmalion had a notion of ‘designation’: an icon was designated when the mouse cursor was within it. Designation had an important effect on focusing the user interface: only visible icons could be designated, and only designated icons could be affected by user operations.

For our purposes the most interesting aspect of Pygmalion was that as systems were programmed by example (using icons) it was impossible to construct systems that separated the structure of the program from its graphical representation. In contrast, HyperCard gives the programmer (but not the user) primitives that can be employed to construct virtual views,* and hence locally circumvent view binding. Nevertheless there was no notion of names within Pygmalion: by design, all active values were directly visible to the user. If Pygmalion had had names, hence environments, view binding would have become available almost automatically.

There have been several attempts at defining two- and higher-dimensional languages. These languages typically do use view binding but their notion of view maybe restricted to

* By program code that explicitly depends on the value of the mouse position and state, or by other methods.

panning over the program text. The aim is generally to present a screen representation of the programming notation, so the imaginary scope rectangles necessarily become actual graphical objects. These notations generally make textual names explicitly visible; the more general idea of view binding is that, although graphical objects are visible to the user, the names bound with those objects are internal (and irrelevant to the end user). Cardelli [3] notes that a two-dimensional notation becomes particularly interesting when applied to two-dimensional data whilst itself being expressible in that data type: he shows that Knuth's operations on boxes [11] can be expressed in his notation.

4. ADVANTAGES OF VIEW BINDING

4.1. Advantages for the programmer

An advantage of view binding is the ability to reuse event names. For example, the event 'mouse down' can be bound in different view scopes to different actions. In one button, it may be bound to a function that performs printing; in another button it may be bound to a function that changes a font. Thus the programmer does not have to invent names for different functions when they are the direct result of user actions or system events. Since naming is one of the hardest parts of programming, and one, if done carefully, which greatly improves the mnemonic value of a program, view binding can be seen as a useful development! In a conventional binding scheme, the same event being able to cause many different actions in different views would have to be associated with a wide range of names, this risks confusion and often results in *ad hoc* and error-prone naming schemes (such as an event name prefixed `W1` is that event originating in window `1`).

View binding, like inheritance in object oriented languages, reduces the usual combinatoric explosion of names; the view scope maps from views to environments and therefore permits a well-controlled overloading of names, to the extent that the number of names bound to user events will be proportional to the maximum number of events (HyperCard has 6 mouse events), rather than the product of events and contexts (HyperCard has seven classes of context, and unlimited instances of them).

4.2. Advantages for the developer

View binding ensures that the bindings appropriate for the current context are immediately available (perhaps merely by changing the mode of the system from 'use' to 'edit'). In conventional systems, the developer would have to work out which bits of a program were associated with which bits of the user interface — the inverse operation to the program's construction of the user interface, mentioned in the introduction.

View binding means that the way that programs are organised corresponds precisely to the way that user interfaces are organised. This is a huge benefit. The correspondence means that simplifying the user interface tends to simplify programs and, conversely, simplifying programs tends to simplify their user interfaces. It means that making a simple change to the user interface (say, we want a button at some position) means only making a simple change to the program and making a trivial, almost negligible change, to its structure. In short, it makes programming user interfaces much easier and far more intuitive: if you know how you want the user interface to be organised, you implicitly know how the program structure should be organised. In HyperCard programs are indeed created *in* the user interface, rather than in some separate process (which probably involves concepts that are unrelated to the user interface, like compilers and linkers), as you organise a user interface you are *simultaneously* organising its program.

4.3. Advantages for the 'power user'

For a power user, a user or organisation, who wishes and is able to customise/enhance the system, the advantage of view binding is that changes to a part of the user interface can be made to the program bound with those views. Typically the power user would change mode to edit, select the view that requires modification, and use an editing tool to modify the program. In conventional systems, it would typically require an expert to locate the piece — or more likely, pieces — of program associated with a particular part of the user interface. Locating the appropriate parts of the program would be unreliable, and would typically involve a deep understanding of the global structure of the entire program.

Example. The Stirling University library bibliographic system has a powerful search facility marred by a poor user interface. Books may be located by words from their titles, but one or two letters may not be used, since single letters are used as commands — thus, a user cannot locate any books about the programming language C. Books may instead be located by their author, but if an author's name is a command, it is impossible to locate. Thus, a user cannot locate a book by the author So, since SO is an abbreviation of the command 'start over,' which backtracks out of the author search! (The library has a manuscript by So.)

Currently librarians are powerless to modify the system and address any of these limitations or bugs.

With a simple user language employing view binding, suitable modifications could be made, and they would (by virtue of view binding) have their effects localised to the points (i.e., view scopes) where the user saw that the changes needed making. There would be no unfore*seen* side-effects, nor would the programmer have to understand any of the (presumably) very large program. Obviously, user enhancement for this example would depend on suitable authorisation.

Even if providing user programmability is unacceptable (say, for commercial or security reasons), note that these user interface bugs arise because the system designers overloaded keystroke streams (e.g., author name and command names are typed into the same-what-should-have-been-a view scope), an error that would have been less likely if their programming environment provided view binding to do the context-sensitive disambiguation for them.

4.4. Advantages for the non-programming user

View binding forces an immediate relation between what can be seen and what functionality is available. View binding ensures this in a way that cannot be guarantied in a program-constructed user interface. Thus, an interactive program implemented with view binding is conceptually consistent.

Before the widespread adoption of windowing systems, the customary approach to extending systems was to add a macro processor. When the user types certain sequences of keystrokes (sometimes using so-called 'soft keys') the value of the associated macro is inserted into the user's text stream. For example, if a user frequently signs himself 'Thimbleby' then a macro processor can easily arrange for typing the function key F8, say, to insert that string more readily. The editor EMACS [18] developed the idea further: any sequence of keys can be bound to a LISP-like function that can then perform any action, whether or not previously available directly from the keyboard, thus overcoming the obvious disadvantage of macro processors that they are unable to extend the functionality beyond that which is anyway already available to the user. With view binding, user events now have a similar standing to user keystrokes except that, importantly, they are disambiguated by application and within that application by its more detailed views.

There are a number of macro systems (e.g., Microsoft's AutoMac [12]) that enable users to record events and to bind them as macros associated with specified applications, but none so far bind by view scopes. In particular, views smaller than whole applications are badly handled, which results in undue sensitivity to the precise positioning of objects — since these macro processors disambiguate events (such as mouse down) by graphical coordinates (which may be accidental when the macro was recorded), rather than by the current view scope which is independent of its physical position!

In short, view binding makes the user enhancement of user interfaces much simpler and more reliable.

4.5. Advantages for user enhanceable systems

There are many reasons why computer systems are not successful, ranging from poor requirements analysis to poor software engineering quality control that results in bugged systems being delivered. One potential solution is ‘user enhanceable systems’ (UES) technology, where the user may modify the system. (‘User’ includes ‘organisation’ and may include — perhaps exclusively — technically competent programmers and support staff. It is a separate issue whether certain classes of user cannot or should not enhance a UES.)

Enhanceable systems technology has many advantages, including:

- better user involvement in the design process;
- wider markets;
- simpler design by delaying commitment [21];
- economies of scale for the supplier — the same system can be delivered to more users, who each enhance the system for their own specific requirements;
- savings in effort for given reliability — a UES is easier to implement reliably than a system that non-extensibly implements many fixed features;*
- designers need not be tempted to add gratuitous features to market a system; instead they will be more concerned with the consistency and coherence of the system.
- the possibility of value added resale markets, with systems being enhanced and resold as bespoke systems.

This is the ideal; however there are problems.

Two serious problems are (*a*) if qualified designers cannot deliver good systems, why should users be any better? (*b*) if users can modify systems, aren’t they are at least as likely to introduce bugs as to ‘enhance.’

The immediate response to both problems is that the designer’s problem is hard: ensuring that many separately specified features are collectively coherent and preserve various properties (such as safety) under all reasonable circumstances is not easy! Users would be no better than designers at coping with this combinatorial complexity. Certainly responsible designers cannot avoid this combinatorial problem, though they may manage it with varying degrees of success. Nevertheless the user is an expert in those very particular and specific circumstances as and when a system is used. Thus with appropriate structuring, factorisation or separation, a user may not need to face the combinatorial design problem of considering all possible circumstances of use. The question is — to paraphrase Dijkstra [6] — whether (and if so, how) such ‘separations’ should be reflected more explicitly in user interfaces.

Many current systems permit users to make trivial enhancements (such as colour scheme changes); more generally a UES must give the user sufficient power to be able to add value to the delivered system, yet be sufficiently constraining so that the user cannot damage the integrity of the system.

View binding provides a very clean mechanism (for graphical systems) to achieve this balance between power and constraint. Specifically, if a designer is to permit a user to be able to enhance certain components of a system, those components must have a visual representation (of a certain kind); conversely, if some aspect must not be modified, it would not have a representation (of that kind). Furthermore, view binding ensures that any enhancements a user makes are necessarily constrained to the scopes in which they appear — a user cannot damage global behaviour of a system unless it is already visible. This is a crucial point if the enhanceable system is safety critical.

The technologies of user enhancement and user interface programming must be contrasted. The design of an interactive system that permits reasonable user enhancement is a difficult design problem in itself. Attempts to make interactive systems easier to implement are not usually concerned with user effort: the aim being to facilitate programmers doing — as part

* Note that compilers are amongst the most complex yet most reliably implemented of systems.

of the continuum of their work — what we here propose users might be able to do. Johnson [9], for example, describes a useful method for programmers that permits decisions about 'look and feel' to be postponed by separating it from semantics. Yet to build a 'clean' UES, the look and feel needs to be carefully considered early. This is arguably a great advantage of view binding!

View binding, as an approach to UES, does not address issues of implementing interactive systems, nor of providing a structure in which view binding itself is effective; these remain difficult problems that must be addressed by other methods. This point may be likened to the relevance of static binding to run time support: it does not make run time support easier. Again, direct manipulation does not make programming user interfaces any easier! Likewise, *designing* interactive systems is not made easier by the structure view binding imposes on the user interface. *Enhancing* such systems, and hence using them over time, is made easier.

5. CONCLUSIONS

There is no novelty in view binding as applied technique. There is novelty, however, in the concepts as such, in the ideal and in the intention. Like direct manipulation [16], view scopes and view binding are concepts to integrate and rationalise features applied imperfectly in some current interactive systems. Direct manipulation is inappropriate for some applications; for other applications, knowing the idealisation helps the interactive systems designer develop a consistent and simpler system. Likewise, view scopes and view binding are not appropriate for all interactive system designs, but they are appropriate for a certain class of system, especially where the user (or programmer) is expected to enhance and customise the user interface.

View binding may be summarised as follows:

- In static binding, program text is visible, but scope is purely conceptual; in view binding, the scope is graphical, but program text is invisible.
- As execution of a program enters (leaves) scopes, the corresponding graphical objects appear (disappear). Conversely, as views appear (disappear) under control of the user, the corresponding program environments are entered (left).
- As a user generates events in graphical regions, the events are interpreted in the program environment of that object.
- Like static binding in a conventional programming language, view binding also restricts the access to names in regions that are 'out of scope.'

The following points are consequences:

- A program can apply (event) names to obtain the corresponding binding (behaviour) from whatever objects are in view. Procedures are functions not only of their parameters and the store, but also of the environments corresponding to their graphical representations.
- View binding provides graphical encapsulation that corresponds directly with program encapsulation, thus ensuring that the behaviour of the user interface is closely linked with the possible behaviours of the program code.
- View binding has a most noticeable effect on program structure, reliability and ease of programming when programs can be modified dynamically, and have unlimited numbers of scopes. Such features are often associated with persistence, which ensures that run time modification to the environment is essentially modification to the program itself.

View binding is a concrete way of controlling, disciplining, *programmable* user interface complexity. As a means for user enhancement it limits the combinatorial design explosion for the user to a reasonable degree. It reduces the impact of bug fixes (actual or attempted) and enhancements (largely) to the view scope in which they are implemented.

Disciplined programmers can always simulate view binding (to any degree) by constructing the user interface in 'the right way': thus they might argue that it is unnecessary.

We argued it imposes and guarantees a useful discipline and structure, and relieves the programmer and user of considerable work — and responsibility. Like all good paradigms, view binding is nowhere mentioned in the program itself, so it imposes no notational overhead in use. Moreover, view binding makes user interfaces more consistent and therefore better understood and more reliably extended by users.

ACKNOWLEDGEMENTS

The following have made very helpful comments on earlier versions of the manuscript: Andy Cockburn, George Coulouris, Brian Gay, Gary Marsden, Maurice Naftalin.

REFERENCES

1. J. ALLEN (1978), *Anatomy of LISP*, McGraw-Hill.
2. S. R. BOURNE (1982), *The UNIX System*, Addison-Wesley.
3. L. CARDELLI (1983), “Two Dimensional Syntax For Functional Languages,” in *Integrated Interactive Computing Systems*, edited by P. DEGANO & E. SANDEWALL, North-Holland, pp139–151.
4. G. F. COULOURIS & H. W. THIMBLEBY (1992), *HyperProgramming*, Addison-Wesley.
5. D. D. COWAN, R. IERUSALIMSCHY, C. J. P. LUCENA & T. M. STEPIEN (1993), “Abstract data views,” *Structured Programming*, **14**, pp1–13.
6. E. W. DIJKSTRA (1976), *A Discipline of Programming*, Prentice-Hall.
7. A. DISESSA (1985), “A Principled Design for an Integrated Computational Environment,” *Human-Computer Interaction*, **1**, 1–47.
8. R. D. HILL (1992), “The abstraction-link-view paradigm: Using constraints to connect user interfaces to applications,” *Proceedings ACM Conference on Human Factors in Computing Systems*, CHI’92, edited by P. BAUERSFELD, J. BENNETT and G. LYNCH, pp335–342, Addison-Wesley.
9. J. JOHNSON (1992), “Selectors: Going Beyond User-Interface Widgets,” *Proceedings ACM Conference on Human Factors in Computing Systems*, CHI’92, edited by P. BAUERSFELD, J. BENNETT and G. LYNCH, pp273–279, Addison-Wesley.
10. M. F. KLEYN & P. C. GINGRICH (1988), “GraphTrace — Understanding Object-Oriented Systems Using Concurrently Animated Views,” *Proceedings Object-Oriented Programming Systems, Languages and Applications* (OOPSLA), pp191–205.
11. D. E. KNUTH (1984), *The TEXbook*, Addison-Wesley, Reading: Mass.
12. MICROSOFT (1988), *AutoMac III*, Microsoft Corporation Document No. 690910001–200–R00–0788.
13. MICROSOFT (1992), *Visual Basic Programmer’s Guide*, Microsoft Corporation Document No. DB27591–0792.
14. B. MYERS & W. BUXTON (1986), “Creating Highly-interactive And Graphical User Interfaces By Demonstration,” *Computer Graphics*, **20**, pp249–258.
15. A. J. PALAY (1992), “Toward An “Operating System” For User Interface Components,” pp339–355, *Multimedia Interface Design*, edited by M. M. BLATTNER & R. B. DANNENBERG, Addison-Wesley, Reading: Mass.
16. B. SHNEIDERMAN (1983), “Direct Manipulation: A Step Beyond Programming Languages,” *IEEE Computer*, **16**(8) pp57–67.
17. D. C. SMITH (1975), *PYGMALION: A Creative Programming Environment*, Stanford University Computer Science Department Report No. STAN–CS–75–499.
18. R. M. STALLMAN (1984), “EMACS: The Extensible, Customisable, Self-Documenting Display Editor,” in *Interactive Programming Environments*, edited by D. R. BARSTOW, H. E. SHROBE & E. SANDEWALL, McGraw-Hill Pub., pp300–325.
19. M. A. TARLTON & P. N. TARLTON (1989), “Pogo: A Declarative Representation System For Graphics,” in *Object-Oriented Concepts, Databases and Applications*, edited by W. KIM & F. H. LOCHOVSKY, Addison-Wesley, pp151–176.
20. R. D. TENNENT (1981), *Principles of Programming Languages*, Prentice-Hall.

21. H. W. THIMBLEBY (1988), “Delaying Commitment,” *IEEE Software*, **5**(3), pp78–86, 1988.
22. H. W. THIMBLEBY (1989), “A Literate Program for File Comparison, in Literate Programming,” *Communications of the ACM*, **32**(6), pp740–755.
23. H. W. THIMBLEBY (1990), *User Interface Design*, Addison-Wesley.
24. P. WISSKIRCHEN (1990), *Object-Oriented Graphics*, Springer-Verlag.
25. S. WOLFRAM (1988), *Mathematica*, Addison-Wesley.

BIBLIOGRAPHIC NOTES

Harold Thimbleby is Professor of Information Technology at Stirling University. He obtained his PhD at Queen Mary College (London) in 1981, in user interface design. His particular concern, illustrated in this paper, is to support the principled *design* of high quality interactive systems, so that they are immediately usable enough or can accommodate easily to users’ requirements. This is in contrast to, but complements, approaches where feedback from use and users helps improve a design. (The latter approach, of course, relies on some sort of design already existing.) Harold Thimbleby is married and has four children.