Painless Panes for Smalltalk Windows

James H. Alexander

*Computer Research Laboratory**
Tektronix Laboratories
Beaverton, OR 97077

ABSTRACT

Current windowing systems (i.e., Macintosh, Smalltalk) give the user flexibility in the layout of their computer display, but tend to discourage construction of new window types. Glazier is a knowledge-based tool that allows users to construct and test novel or special purpose windows for Smalltalk applications.

The use of Glazier does not require understanding Smalltalk's windowing framework (Goldberg, 1984; Goldberg & Robson, 1983). As a new window is specified, Glazier automatically constructs the necessary Smalltalk class, and methods (programs). Windows are interactively specified in a Glazier window - the user specifies type and location of panes through mouse motions. Panes can contain text, bit-maps, lists, dials, gauges, or tables. The behavior of a pane is initially determined by Glazier as a function of the pane type and related defaults. These default behaviors allow the window to operate, but do not always display the application information desired. In that case, the user can fix the window's behavior by further specification. Such alterations require only knowledge of the application, not of the windowing system.

Glazier allows the prototyping and development of full-fledged Smalltalk windows, and allows a flexibility that will change window usage in two ways. First, it will allow end users to construct special purpose windows for viewing data from an application in manners unanticipated by the system designers. Second, system developers will be encouraged to prototype and evaluate many window configurations before settling on a final choice. Both alternatives will result in windows that are more satisfying to the end-user.

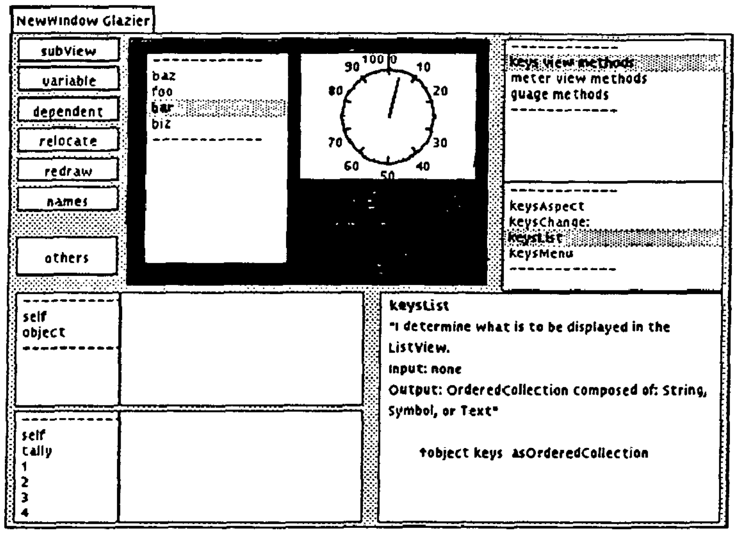
*The author's current address is: U S WEST Advanced Technologies, Science and Technology, 6200 S. Quebec St., Suite 170, Englewood, CO 80111

The makeup of Smalltalk or Macintosh-style windows is typically viewed as a fixed component of the computer interface. Windows are provided to the end user by the system designer and cannot be customized. Sadly, users are not allowed the flexibility of their window contents that windows allow for display contents. Thus, the user is forced to use windows that may not precisely fit the needs for his or her use of the application.

Of course, the option of adding new windows is available to some users. A *skilled* Smalltalk user can construct a special-purpose window in an afternoon. Completion of such a task requires detailed knowledge of Smalltalk's model-view-controller (MVC) paradigm (Goldberg, 1984; Goldberg & Robson, 1983). This is perceived as an inconvenienct, tedious task and is hardly something a novice Smalltalk programmer can or should attempt.

This paper discusses Glazier, a tool that encapsulates knowledge about building Smalltalk windows, and assists a user in developing new Smalltalk windows. Glazier works as an assistant, relieving the user from the burden of thinking about windowing details. Instead, the user needs only to understand how to operate the data structures for the application being displayed by the window. Window development now becomes a symbiotic process, Glazier provides the knowledge on how to build the window and the user provides knowledge about how the application is used and how the window should behave.

There are a numerous other systems to support interface development in a like manner. Bass (1985) describes a system for developing VT100 style interfaces on top of base-level applications. The system supports a wide range of user needs, but cannot be configured dynamically by the user. The Trillium System (Henderson, 1986) supports prototyping of copying machine interfaces and



```
keysList
"I determine what is to be displayed in the
ListView.
Input: none
Output: OrderedCollection composed of: String,
Symbol, or Text"

    ↑object keys asOrderedCollection
```

Figure 1 - An example GlazierView used for constructing new windows. A prototype of the new window appears in the proto-window, the top-center subView of the GlazierView. The seven buttons on the top left of the window are used in adding and modifying panes within the proto-window. The two subViews on the bottom left are used for inspecting objects in the application the new window will display. Finally, the three subViews on the right are used for accessing methods in the class controlling the operation of the new window.

allows designers to build and test control panels. Other user interface management systems support construction of front ends for applications (Hayes, Szekely, & Lerner, 1985). None of these systems, however, has provided the user or developer with a dynamic environment for building generic windows. Glazier allows a user to build a wide range of window types, and use them as they are being built.

This paper will discuss the operation of the Glazier, the method for constructing windows, and finally the implications of this new window construction technique.

1. Overview of Glazier

When a user develops a window with Glazier, the result is a new Smalltalk class that defines the operation of the new window. This class is written by the Glazier, according to a standardized style, as the user specifies the window. The user constructs a new window with the help of a special purpose window, a GlazierView. This window is opened by a parameterized message which passes a pointer to an instance of the application which the new window will be designed to display. In the GlaizerView the user can dynamically specify and test a new window. Figure 1 shows a picture of a GlazierView. This view has four general panes to assist the user with the construction of the new window:

- *The new window pane* - The top center pane of the GlazierView is a prototype of the new window. This proto-window operates exactly as the new window will operate when opened. Initially the proto-window is empty and gray. As panes (subviews in Smalltalk terminology) are added to the new Window, they are displayed as they would be in an ordinary Smalltalk window. The user can operate the proto-window and test it's functionality. At any time, the user can open a stand-alone version of the new Window and it will operate exactly like this pane in the Glazier.

- *Buttons* - used for creating a new pane, adding a new instance variable, creating a new dependent, relocating a subview, refreshing the proto-window, displaying the names, and a button for getting a menu of other available commands.

- *Data inspectors* - Two of the GlazierView's panes are object inspectors that allow the user to examine the values of instance variables in the proto-window instance and the application proto-window is designed to display.

- *Code browser panes* - Three panes represent a code browser on the class being generated. One to show the protocol categroies for new code generated by the Glazier (see below), one to show the selector names of a protocol selected in the previous pane, and one to show the method selected. In these panes the user can modify and augment the code automatically generated by Glazier.

1.1 Specification of a new window.

To specify the contents of a new window, what needs to be specified is the size, location, type and behavior of the panes in the window. The developer describes the size, location and type of pane; Glazier provides default methods for the behavior, which the user can modify to suit specific needs.

The user initiates the addition of a new pane to the proto-window by pressing the button labeled: `subView`. First, Glazier will prompt the user for the name of the new pane. At this point, the user specifies the size and location of the pane by indicating a rectangle the pane will occupy. After the area is specified, Glazier will present a menu of the types of panes supported and the user can select the desired type. Supported panes are currently: FormViews (bit maps), TextViews, ListViews, TableViews, StatusViews, and ButtonViews. The pane is added to the window and code describing the operation of the window is written, compiled in the new window's class and displayed in the GlazierView's code browser panes. This code is a set of short modular programs called methods (according to the Smalltalk convention). New methods are given selector names composed of the pane name and the generic function name spliced into a single word.

The methods generated by Glazier are default methods describing behavior necessary for operation of the pane. Typically the defaults will not operate exactly in the manner desired by the user. It is up to the user to change these methods to produce the behavior desired. The default methods are very stereotypic, and the user changes are usually regarding data is accessed by the new window class. Such changes do not require knowledge of pluggable views or the MVC framework, only familiarity with the application supporting the underlying object.

The user can add as many subViews to the new Window as are needed. In addition to simplifying the addition of subViews to a window, Glazier performs numerous other functions that facilitate the construction of windows:

- *Addition of variables* - often the new Window class needs to keep track of selections in various panes, history information or other types of information. The user can touch the variable button and add an instance variable to the new class. Glazier asks the user to enter code describing the initialization routine of this variable. The code is added to the initialization procedures for the new window.

- *Creation of dependencies* - When a instance variable value changes in a manipulator, it is often appropriate to update a corresponding pane in the window. The user can indicate such variable/pane dependencies by touching the dependent button. Glazier will present the user with a menu of all the instance variables, and after selection of one will ask the user to point to the pane that is dependent upon that instance variable. This dependency is stored in the code for the new window class.

- *Relocation of panes* - The user can experiment with multiple pane placements by touching the relocate button (indicating the pane to be moved) and framing the new location. Any pane may be moved at any time. Panes often need to be relocated according to specific values and Glazier provides two other means of relocation: automatic relocation using an algorithm, or explicit user input of exact coordinate locations.

- *Other actions* - Other functions provided will allow the user to examine all aspects of the new subview and manipulate it by hand. In addition, there are functions for removing subviews, instance variables, and dependencies.

2. How Glazier builds a window.

Building a Smalltalk window involves the creation of a MVC triad that controls the presentation of object data in a window. A MVC Triad is pictured in Figure 2. The Glazier takes advantage of standardized Views and Controllers available in Smalltalk and builds windows by helping the user generate a new Model. Thus the types of windows generated through Glazier is limited by the views and controllers available in the Smalltalk system. Because of the conventions built into some Smalltalk Views (specifically pluggable views, explained below) the construction of models

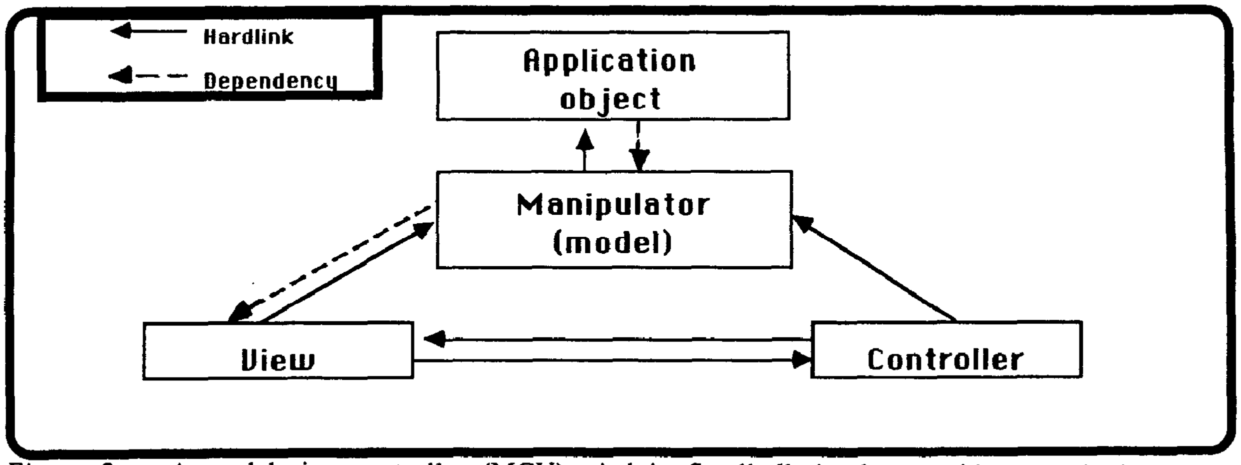


Figure 2 - A model-view-controller (MCV) triad in Smalltalk is shown with a manipulator communicating with the application. The manipulator provides modularity for the application code as well as the view code. Typically, the manipulator keeps track of selections in the window, the relationships between panes within a window, and the relationship between window panes and components of an application. In most cases, communications is achieved directly by passing messages to instance variables, however communication from the application object to the manipulator and from the manipulator to the view is achieved through dependencies.

can be very stereotypical, enabling a mechanized process. To understand this process, however, one must understand some about the MVC framework. This will be discussed briefly.

2.1 A MVC Primer

Smalltalk divides the responsibility for window management among three types of objects:

- *The Model* - an object representing the data structure of the application, and contains or can access information to be displayed in the window. The model acts as an agent between the view/controller pair and the applications object, and communicates information. These agents are called browsers, or manipulators.* Formerly, the application objects themselves served as the model. However this practice forced customizations of the object which hinder the reusability of the code.

- *The View* - an object that controls the visual aspect of the window. In its simplest form a view keeps track of the bounding box of the window, and the means for updating the contents of the window. Views may be composed of subViews. For clarity's sake in this paper top-level views will be referred to as windows, and views within a window will be referred to as panes.

- *The Controller* - an object that controls the user interaction with the window. In its simplest form, it determines if and where the pointer is contained within the window, it describes what happens when a window becomes active, and it determines if the user has pressed any keyboard or mouse buttons.

*To distinguish them from the Smalltalk system browser, a standard Smalltalk window, I will call them manipulators.

Smalltalk has classes of Views and Controllers sufficient to fit the needs a programmer might encounter. Generally, views and controllers are developed as matched pairs intended to be used together and only with each other. For example, the Standard System View class and the Standard System Controller class are always used together and, for our purposes, can be thought of as a single entity. From here on the term View will be used loosely to refer to a view/controller pair.

2.1.1 Pluggable Views

Pluggable-views standardize the interactions between views and models. This is done through *adaptor messages* which are messages implemented by the model (manipulator) which the view should send to indicate the need for information or action. These selector names are held in an instance variable of the view and executed when appropriate. Some common adaptor messages are:

- *aspect* - sends the view the proper data representation for the view. This will be different for each pane in a window. For example a checkbook object might want to send the balance to one pane, a list of outstanding checks to another pane, and a graph of the cumulative balance over the past month to a third pane. The checkbook's manipulator would implement one aspect message for each of these panes in order to provide the proper data.

- *change* - activates whenever something is changed in the view. For example, a button might be pressed or a text item changed. The parameter would communicate to the manipulator the value of the change, and typically prompt some action in the application object. In the checkbook example this could be the action of updating the balance when a check is written.

- *menu* - A method implementing the menu for the view.

Adaptor messages place the responsibility for customization of view on the manipulator rather than the view or application data object. Thus, the standard collection of views and controllers can be plugged into varying models without modification. All that changes is the adaptor message held in the appropriate adaptor message slot.

2.1.2 **Building a manipulator for pluggable views**

The manipulator stands between the view (and controller) and the application object(s) to be displayed by that view. The manipulator communicates to the view appropriate information about displaying various aspects of the object, and communicates to the application object information about what changes should be made to it. A window may be composed of one or more panes of various types, with each pane showing different perspectives of the same object. It is convenient (but not necessary) to have the one manipulator control all of these panes and coordinate their behavior.

When a programmer defines a window, s/he must define three factors:

- The type of each subView composing the window (i.e., Boolean View, FormView, TextView).

- The location of each subView within the window.

- The behavior of the subViews as defined by the adaptor messages.

Manipulators typically look like simple state machines. Creation is a straight forward, but time consuming process. One needs to define the type and location of all views that will compose the window. For each of these views, the proper aspect methods need to be generated. Typically, one should build an instance variable into the manipulator for each of the views to keep track of selections in the view.

The prime benefit of the manipulator is modularity. If a window is coded without the use of a manipulator the windowing code must be part of either the view class or the application class, thus reducing the generality of both these classes. By using the manipulator, the developer can keep the generality of the view classes and get customization though the new Manipulator class.

2.2 **What Glazier does.**

All Manipulators share a similar form, yet they do not share the type of similarity that can be shared through Smalltalk inheritance. To inherit something in Smalltalk, either the structure of a class or the messages they respond to should be consistent. Manipulators do not share structure and do not respond to the same set of methods. Rather, they have numerous methods that have the same form, but differ in the selector name. For example, a manipulator may have two or more *foo*Aspect methods all of which have the same form, but different values replacing *foo*. Glazier's knowledge base allows the programmer to share knowledge about the structure of these regularities.

Glazier's knowledge base consists of:

- a message for opening each type of pluggable view in a Smalltalk System

- default methods for the adaptor messages required by the pluggable views.

- updating messages for each of the window types*

*This knowledge base is a reflection of the lack of a standard to propagate changes in the ST-80 system. A more consistent updating message convention would obviate this knowledge.

This knowledge base is contained in a class called Manipulator. All manipulators created by the system are implemented as subclasses of Manipulator. A subclass of Manipulator, inherits code that standardizes the operation of the new manipulator type, but the class includes no information about the new manipulator or the window to be controlled by this manipulator. This code is generated in the process of building a new window.

The four types of information that is added to the new Smalltalk class when panes are added within a window are:

- *Pane types* - the name and type of each pane
- *Pane locations* - an association of the name and it's location within the window (in relative coordinates)
- *Instance variable dependents* - which subviews are dependent upon the instance variables defined by the subclass. Through the use of this information, when an instance variable value changes, the appropriate panes in a window can be changed accordingly.
- *Adaptor messages* - specific messages for each of the panes are generated. These messages are implemented according to defaults appropriate for the type of pane being generated.

Unlike normal Smalltalk panes, each pane of a window has a name attached to it. This gives the system a handle for the generation of the pane and the corresponding adaptor messages. In Glazier, the adaptor messages are prefixed by the pane name. a new adaptor message set is created for each and every pane in the window.

3. Correspondence of Glazier generated windows with normal windows

Glazier was not intended to build windows that do everything that the programmer can imagine. Because of this, one of the design goals behind the Glazier was to continually produce code that is human readable and conforms to standard Smalltalk style. To some extent the system both failed and succeeded on this count.

The adaptor messages written by the Glazier are typical Smalltalk code. They are simple methods, and are commented as to their function, inputs, and outputs. The method names (selectors) are the pane name concatenated with the adaptor message name. The automatically generated code is standardized and easier to read than code normally found in a programmer developed system.

The main departure from current Smalltalk style is in window opening methods. Normally a window is created with code resembling the following:

```
open
  | topView subView|
  topView_StandardSystemView new.
  subView_SelectionInListView new.
  topView addSubView:subView
          in: (0@0 extent: 1@1)
          borderWidth: 2.
   topView controller open
```

The opening methods actually contain the explicit specification of the size, location and type of pane to be added to the window. The Glazier separates this out by holding the pane types, locations and sizes in a data structure (actually in a method that regenerates the data structure). The opening method for all Glazier developed windows looks like:

```
open
  | topView subView |
  topView_StandardSystemView new.
  self addSubViewsTo: topView.
   topView controller open
```

With this aproach, `addSubViewsTo:` controls all of the pane addition by consulting the appropriate data structures. This departure from the standards is not critical, though it may be unexpected. In fact it appears to be a convenience that should be added to all window opening methods. By keeping the size and location of panes out of hard coded methods, one is afforded a flexible window framework. Using this convention it becomes a simple matter to change and save the size and location of panes in windows.

4. Summary

4.1 Benefits of Glazier

Glazier should have a revolutionary effect upon the construction and use of windows in computer systems. Instead of being fixed interfaces to an application, windows can now be experimented with and manipulated for maximum efficiency, and for special purpose functions. One can now imagine a system designer experimenting with various pane configurations before settling upon the one that will be delivered with the system. One can also imagine a user constructing a special purpose window for a particular application. Suppose a person needed a special-purpose window to show all of the checks in a checking account that were above a certain dollar amount. A user could quickly set up a window with a list and a dial in which the setting of the dial represents the cut off value and the list contents are all of the checks above that value.

One style of window construction encouraged by Glazier is a style in which the panes do not consume all of the internal space of the window. Thus, the window may have blank space between the edges of panes. After working with these for a while, I have become convinced that empty space in a window design is at least as important as sagacious use of white space in the layout of a book or letter. Empty space can be used to guide the eye to related areas within the window.

4.2 Dependent updating in the Glazier

One of the most confusing components of building a window in Smalltalk is the propagation of change messages to dependent panes. Typically, a view is designated as a dependent of a particular object, and whenever the object is changed that view is updated. However there are two problems with this scheme for our purposes:

- All dependents are updated, causing unnecessary delays
- If something is made a dependent of an object and that object is replaced instead of updated, the dependents cannot be updated.

Glazier allows a more modular updating mechanism. This mechanism is not particular to Glazier but has been developed in parallel. With the new mechanism views can be designated as dependents of the *instance variables* of an object rather that the object or the values held by the instance variables. Thus, whenever any change is made to the instance variable a change message can be issued updating only those panes dependent upon that instance variable. This causes faster overall window response than announcing the general object change and makes windows more responsive to user inputs.

5. Future work

Glazier is still at a simplistic stage of development, it builds the default adaptor methods only as a function of the type of pane they are constructed for. This means that the default is typically the greatest common denominator between the possible objects to be displayed. For example, the meter panes require the display of some number. The default method chosen to display this number must be implemented by all classes in Smalltalk. One of the few choices that does this is `size` which has little meaning to the user for most objects. The next generation of Glazier will have knowledge on how to write the methods according to the type of data to be displayed. This would result in default methods that would be more likely to be appropriate upon first creation.

Likewise there are other conventions that should be added to extend Glazier. ListViews, for example, often require a special instance variable in the manipulator to keep track of the item selected. Also, FormViews usually require a variable for caching the form to be displayed. It may be reasonable for Glazier to automatically generate an instance variable for each new pane added to the window, and this may be done in the near future.

In addition, Glazier should be expanded to construct more and more types of windows. Currently, Glazier only supports panes with limited intercommunication. Also, it is not possible to support panes defined by the user. Consequently, Glazier constructed windows are monolithic, in that there is only one level of pane depth for a window. Experience has shown that it is more convenient to have panes within panes, and this is another obvious extension for the functionality.

Finally, the type of code sharing implemented by Glazier suggests many different applications not possible with libraries or the inheritance mechanism of Smalltalk. Glazier is a repository for the knowledge of how to construct Manipulators. By codifying and automating such knowledge it is possible to share knowledge on how to build regular structure. This could be of use in many areas. For example, two of the most regular structures in Smalltalk-80 are Views and Controllers. One can imagine having two more Glazier-like entities one for constructing views and one for constructing controllers. Such possibilities are being explored.

6. References

Bass, L. J. A generalized user interface for applications programs (II). *CACM*, 1985,28,617-627.

Goldberg, A. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley Publishing Company, Reading, MA: 1984.

Goldberg, A. & Robson, D. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Publishing Company, Reading, MA: 1983.

Hayes, P. J., P. A. Szekely, & R. A. Lerner. Design alternatives for user interface management systems based on experience with Cousin. *CHI'85 Proceedings*, 1985, 169-174.

Henderson, D. A. The Trillium user interface design environment. *CHI'86 Proceedings*, 1986, 221-227.

7. Acknowledgements

I owe thanks to numerous people for the help they provided on this project. Principally I'd like to thank Jeff Staley, Ward Cunningham, Allen Wirfs-Brock and Jeff McKenna for their suggestions.