Creating Host Compliance in a Portable Framework: A Study in the Reuse of Design Patterns

Phillip M. Yelland

Affiliation: ParcPlace-Digitalk Inc.

** Author's current address: JavaSoft (Sun Microsystems Inc.), 2550 Garcia Ave., UCUP01-202, Mountain View, CA 94043-1100; Phillip.Yelland@eng.Sun.COM.*

Abstract

This report describes an experiment carried out at ParcPlace-Digitalk which sought to increase the look-and-feel compliance of portable applications built using the company's Smalltalk-based VisualWorks product. We outline the structure of the current VisualWorks user interface framework, and the precise requirements which the experimental system sought to fulfill. We go on to show how we were able to reuse design patterns from the literature in a *generative* fashion, to direct the evolution of the new framework. This contrasts with most pattern-related work to date, which has concentrated on discerning design patterns in existing systems. Finally, we draw generalizations from our experience concerning the evolution of software architecture using patterns.

1. Introduction

The VisualWorks product from ParcPlace-Digitalk is a Smalltalk-based program development environment intended to facilitate the production of graphical client-server applications. One distinguishing feature of VisualWorks is the very high degree of portability it offers; applications developed using VisualWorks run without alteration on any brand of Microsoft Windows, the Macintosh, OS/2 and several varieties of UNIX. ** In fact, VisualWorks applications also enjoy a unique form of snapshot portability, which allows a running application to be shut down on one platform and restarted in the same state on another.*

VisualWorks comprises an extensive class library, containing a number of complementary application frameworks designed to support different aspects of client-server development. In this report, we describe an experiment which involved one of these frameworks - that used in the development of graphical user interfaces. Currently, VisualWorks' user interface framework achieves portability by relying exclusively on *emulation*; every element of an application's user interface is drawn and animated in Smalltalk, using a number of platform-independent I/O primitives. Unfortunately, this approach imposes a number of penalties on VisualWorks applications, particularly as regards host look-and-feel compliance - the degree to which applications conform to the look-and-feel guidelines stipulated for a particular platform[A87] [M91]

implementation of the user interface framework so as to incorporate controls * *Throughout this report, we use the terms control and widget interchangeably to denote user interface elements such as buttons, lists, input fields and the like.* provided directly by the host.

A number of software design patterns taken from the literature played a pivotal role in the development of the new architecture produced during the experiment, and their use is described in detail in the main body of the report. Documented experience of this so-called *generative* [C95a] [DeB95] use of design patterns is fairly rare, particularly in application to the evolution of existing frameworks. * *[S95] is amongst the few examples.* Given the relative infancy of the field, most pattern-related work to date has concentrated on discerning design patterns in existing software architectures.

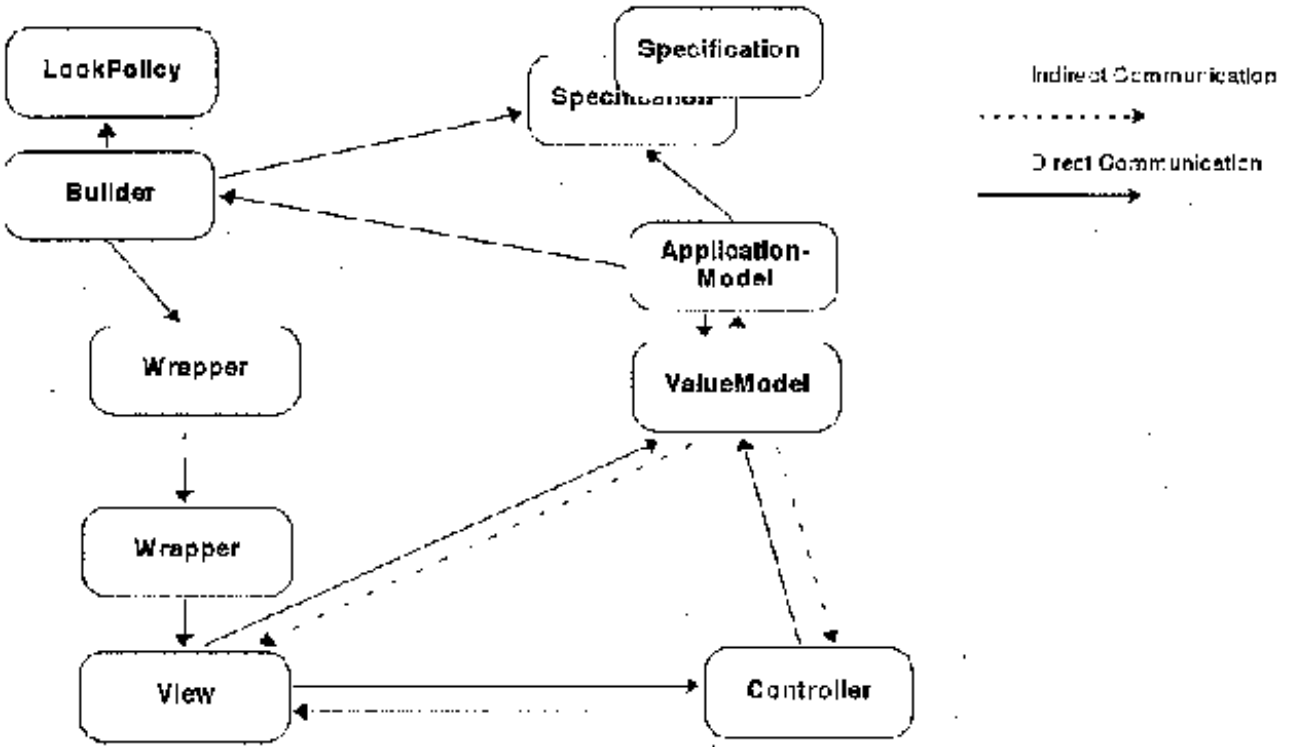


Figure 1. The existing VisualWorks user interface architecture

The next two sections sketch the architecture of the current VisualWorks user interface framework and describe in more detail the shortcomings in it which the experimental architecture sought to address.

2. The Existing VisualWorks Architecture

Figure 1 is a greatly simplified rendition of an application built using the VisualWorks user interface framework. ** For more details, see [P94].* The ApplicationModel ** All Smalltalk class names are rendered in san-serif.* depicted in the diagram encapsulates the application-supplied parts of the interface; it is responsible for specifying the form and layout of windows and controls in the interfaces, and for supplying the information to be displayed.

The specification of an interface in VisualWorks takes the form of a collection of Specification objects, which define the characteristics of the interface in a platform-independent fashion. To create the interface, the ApplicationModel passes the specification to a Builder object, which traverses the Specification objects, constructing the interface as it does so. In order to render the interface appropriately for a given platform, the Builder is equipped with a LookPolicy, responsible for providing the Builder with components for that particular platform.

Interfaces in VisualWorks are constructed using the Model-View-Controller pattern, which VisualWorks inherits from its predecessor, Smalltalk-80. Thus on Windows, for example, a button is realized using Win3ButtonView (responsible for displaying the button), a ButtonController (which handles user input) and a ValueModel. The latter is a standard form of model object that acts as a conduit between the ApplicationModel and the View and Controller. The View associated with a control may be supplemented with one or more Wrappers, which provide decoration and/or additional capabilities, such as borders or scroll-bars. The ApplicationModel is able directly to access the Wrappers, Views and Controllers in the interface through the Builder, to which it retains a reference after the latter has finished constructing the interface.

As we pointed out in the introduction, VisualWorks makes no attempt to make use of controls provided by the platform. Instead, the system emulates all the controls in the user interfaces it produces. Hence, the Win3ButtonView alluded to above depicts a Windows 3.1 button using only graphical primitives furnished by the Smalltalk Virtual Machine [GR83], and a ButtonController effects the appropriate behaviour based on low-level events originated by the Virtual Machine. This approach gives the system great flexibility and control, and makes achieving portability very easy, since the Virtual Machine's I/O facilities are platform-independent. However, it carries with it a number of liabilities, as we describe in the next section.

3. Requirements for Evolution

The following are some of the notable weaknesses of VisualWorks' emulation-based approach to user-interface construction:

- Faithfully rendering a given platform's look and feel by emulation can be a demanding task; there are a number of respects in which current VisualWorks applications exhibit less than complete fidelity on certain platforms. Furthermore, a change in platform look-and-feel (such as that which occurred with the introduction of Windows '95, or Macintosh System 8.0) means developing a new set of emulated controls. Both of these problems can be overcome by making use of platform-provided controls, rather than do-it-yourself emulations of them.

- The apparent responsiveness of a VisualWorks user interface is critically dependent on the speed of the emulation code, which is written almost entirely in Smalltalk. On a less-than-well-endowed machine, this can result in VisualWorks applications appearing unacceptably sluggish. Performance can be improved considerably by using host controls.

- The platform-independent input/output mechanisms used by the emulated VisualWorks controls differ markedly from those employed by host controls. The differences make it difficult to achieve inter-operation with platform user interface elements, such as Windows '95 Common Controls [C95b], OS/2 2.0 Controls [H92], in-place-activated OLE applications or OCX's [M94b]. It can also prove frustrating for developers who expect to apply to VisualWorks applications the same techniques they are accustomed to using with other applications on a particular platform.

In alleviating the problems listed above, we were determined not to vitiate the strengths of the current VisualWorks product. This led us to the following list of requirements for the new VisualWorks user-interface architecture:

R1. It should make use of platform-provided controls and I/O mechanisms wherever possible.

R2. It should increase the accessibility of platform facilities from VisualWorks.

R3. It should have minimal impact on existing VisualWorks applications; ideally, applications written for the emulated VisualWorks user interface framework should operate without alteration on the new platform-control-based framework.

R4. It should retain the portability (of both code and snapshots) of VisualWorks applications.

R5. It should impose no penalties in terms of application speed or memory requirements, compared with the existing emulation framework.

R6. It should be easier to use, maintain and enhance than the emulated framework.

In the next section, we introduce the design patterns that were central to the accomplishment of these goals.

4. Evolution Using Patterns

To recapitulate a point made briefly in the introduction: Given that systematic descriptions of software design patterns have only been available in significant numbers relatively recently, it's no surprise that most work in the area has been of a largely descriptive nature. The authors of[G+95], for example, make it clear that their intention is to systematize existing practice by identifying

patterns in existing applications. By contrast, we found when embarking upon our experimental revision of the VisualWorks user interface framework, a sufficient body of work had built up that we were able to make use of already documented patterns in a *generative* fashion [C95a][DeB95] to guide the development process. This allowed us to design our new architecture by identifying those patterns best suited to fulfilling the requirements placed upon it. In fact, the entire exercise produced only one truly novel pattern (the ACCUMULATOR ** For clarity?s sake, the names of patterns are rendered in SMALL CAPS.*, described on page 9). All the remaining patterns employed were taken from the literature, either directly or by adaptation. We would represent this not as evidence of a singular lack of originality, but as a ratification of the oft-repeated assertion that design patterns constitute viable candidates for reuse.

Pattern	Manipulation	Requirements Cited
1. Layered Architecture	Added	R1, R2, R4, R6
2. Adapter	Added	R3
3. Interpreter	Retained	R3, R6
4. Decorator	Removed	R6
5. Model-View-Controller	Modified	R1
6. Bridge	Added	R1, R4
7. Proxy	Added	R1, R2
8. Self-Addressed Stamped Envelope	Added	R2, R6
9. Flyweight	Added	R5
10. Accumulator	Added	R2, R6

Table 1: Design patterns in the evolution of the experimental system

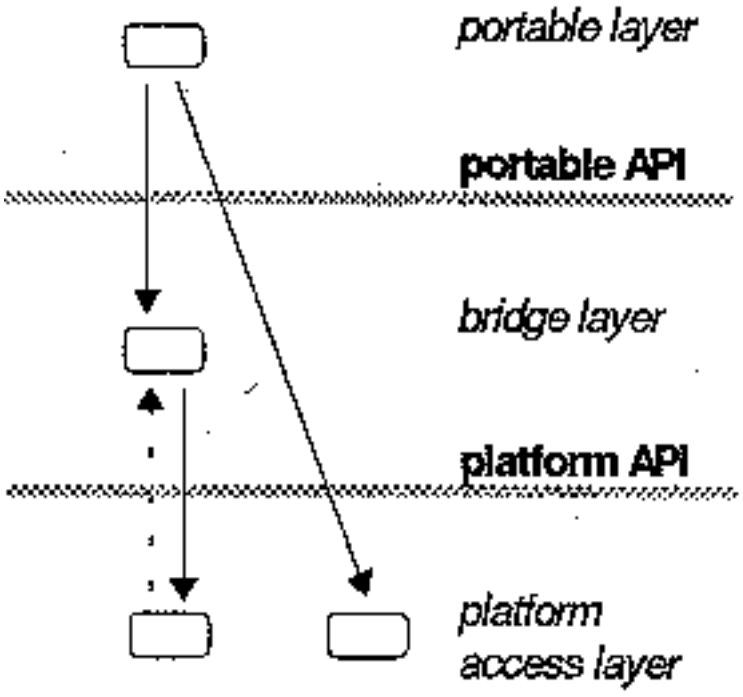
In the remainder of this section, we describe the design patterns which figured in the evolution of the new architecture, and the ways in which those patterns were manipulated in response to the requirements set out above. Note that the architectural modifications did not consist solely of pattern additions - some patterns identified in the old VisualWorks architecture were removed or substantially modified. A brief summary of this section's discussion is given in Table 1.

Space considerations preclude the formal presentation of each pattern. Instead, we will simply characterize them in general terms, and direct the reader to the references for further detail. Unless otherwise indicated, a formal description of each pattern may be found in [G+95], and for the sake of brevity, we will usually refrain from referencing [G+95] explicitly in the following.

Where appropriate, illustrations are provided of the application of particular patterns; a complete depiction of the new architecture - from which these illustrations are excerpted - may be found on in Section 5.

Pattern 1:

Layered Architecture



A system with a LAYERED ARCHITECTURE [B95a] is constructed as a number of linearly-ordered tiers, with the behaviour of objects in a given layer normally expressed in terms of the behaviour of objects in the same or lower layers. Objects in lower layers are often said to provide *services* to objects in higher layers.

Layers were a feature of software designs for many years before the use of object-oriented programming technology became widespread [P72] [T88]. The application of layers has been particularly prominent in the design of a number of portable systems (possibly the most well-known recent example is Windows NT [C94]). Such systems contain one or more platform-dependent layers which are responsible for presenting a platform-independent set of services to higher layers in the system. This allows applications inhabiting the upper layers of the system to be written in a portable fashion.

The user interface framework developed in the experiment uses a version of the layered pattern to reconcile support for the development of portable applications with the provision of direct access to platform facilities. The new architecture divides objects into three layers: In the top-most layer are *portable* objects, which are able to draw on the services provided by two lower layers. The middle layer - called the *bridge layer* for reasons explained below - supplies a platform-independent API to the portable objects using capabilities drawn from the bottom-most layer. This bottom-most layer - the *platform access layer*- presents a rendition of the platform's own API (more details later in the report).

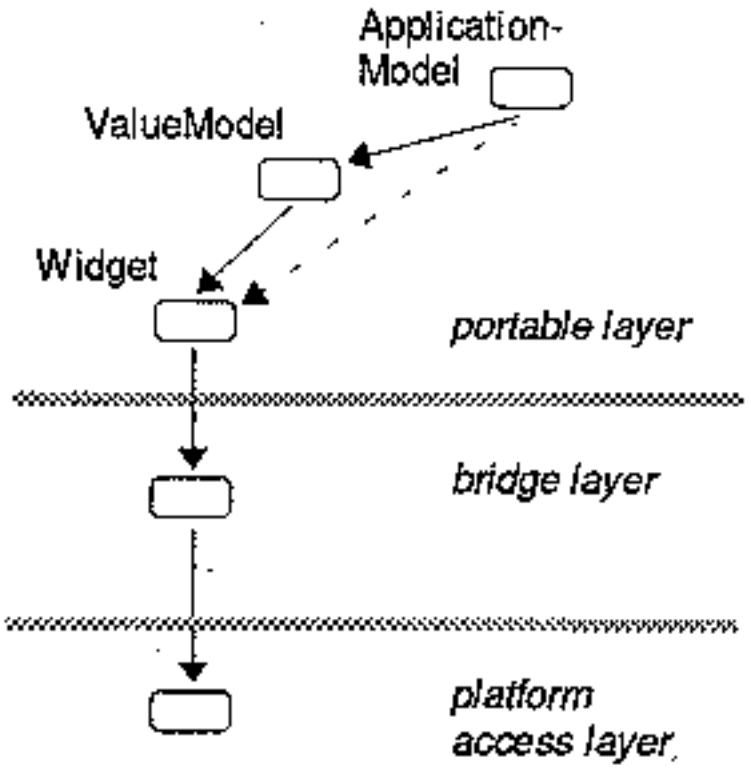
This architecture allows the application developer a choice: write entirely to the

This architecture allows the application developer a choice: write entirely to the platform-independent API provided by the bridge layer ** Or more precisely, use system-provided portable objects which use the bridge layer's API - see pattern 2.* to produce a portable application, or use the platform access layer to write a non-portable application which takes advantage of platform-specific functionality.

A layered architecture also allows the new framework to retain the snapshot portability of its predecessor, because the system is structured in such a way that all the state of an application's user interface which needs to be present in a snapshot (so that it can be reincarnated on the same or a different platform) resides in the portable layer; no persistent state is kept in the bridge- or the platform access layer. Therefore, taking a snapshot of a portable application is simply a matter of recording the state of the portable objects it comprises. ** Of course, applications which make direct use of objects in the platform access layer need to make special provision for snapshots; the system supplies facilities to support this.* When the snapshot is regenerated, the objects in the bridge- and platform access layers are appropriately reconstituted, reproducing the original user interface using controls from the new platform.

Pattern 2:

Adapter



The chief participant in the ADAPTER pattern is an object which provides an interface to a client based upon the services provided by another object. [G+95] observes that this pattern is most commonly used to match the interface of an existing class to the needs of a new set of clients.

In the experimental architecture, we used the pattern in the converse sense; to present an interface on new classes for use by existing clients. Here, the existing clients are legacy VisualWorks ApplicationModels and ValueModels, written to the portable API formed by the message protocols of the Views, Wrappers, Controllers and Builder in the original VisualWorks system. The new classes are the objects in the bridge layer, which present a portable abstraction of the host. Most of the adapter objects are called *widgets*, since they represent VisualWorks' portable user interface controls.

This use of the ADAPTER pattern in the experiment was particularly successful; a large proportion of the legacy VisualWorks applications sampled in the project operated correctly with the new user interface framework with little or no modification. Also, since the widgets and their interfaces resided entirely in the portable layer, any legacy application was ensured the same code and snapshot portability with the new framework that it enjoyed with the old.

Pattern 3:

Interpreter

The combination of Builder and LookPolicy described in section can be regarded as an instance of the pattern INTERPRETER composed of instances of two other patterns - BUILDER and ABSTRACT F ACTORY. Together, the Builder and LookPolicy "interpret" the language comprised by the Specification objects, constructing a graphical interface as they do so.

Such is the flexibility of this configuration that we were able to retain both it and the existing implementation of Builder objects in the new framework. It only required the provision of a new LookPolicy to cause the system to construct user interfaces according to the new architecture rather than the old one.

Retaining the Builder had a number of advantages:

- It relieved us of the need to implement a replacement for the class Builder and the ancillary classes associated with it.

- It automatically preserved a major proportion of the API used by legacy VisualWorks applications (recall from section that an ApplicationModel's prime means of access to the interfaces it controls is via a Builder object).

- It allowed the new architecture to coexist with the old emulation system, and with other VisualWorks-based user interface frameworks (such as that used in ParcPlace-Digitalk's VisualWave Internet-based product [P96]), which also provide their own LookPolicy.

Pattern 4:

Decorator

The Wrappers described in section are instances of the DECORATOR pattern - a pattern used to attach additional capabilities to an object dynamically. They are used in the current VisualWorks user interface framework to add scroll-bars, borders and the like to emulated controls.

The experimental architecture dispenses with Wrappers, for the following reasons:

- Wrappers are not entirely transparent to the application developer, who must take care to "drill down" through the appropriate number of wrappers before performing certain operations. For the same reason, the presence of wrappers can also greatly complicate debugging.

- Most of the classes of Wrapper in the emulated VisualWorks framework are required because they provide the implementation of the capabilities they endow. For example, a BorderedWrapper actually draws a border of a given style around its component. Most of these adornments are provided by the platform-provided controls themselves, which eliminates the need for Wrappers to implement them in the new system.

Pattern 5:

Model-View-Controller

The Smalltalk MODEL-VIEW-CONTROLLER (MVC) configuration [KP88] is one of the most venerable of the design patterns associated with object-oriented programming. Many user interface frameworks profess to owe it inspiration - examples include the Microsoft Foundation Classes [M94a] and MacApp [A89]. However, very few of these frameworks make use of the configuration in the original form embodied in Smalltalk-80; almost without exception, they conflate theView and Controller into a single object. The main reason for this is that Controllers in Smalltalk-80 (and latterly, in VisualWorks) are largely an artifact of the implementation of control-flow in the (emulated) UI. In a user interface based upon the use of platform-provided controls, a distinct implementation of control-flow is unnecessary - and actually very difficult to achieve, since the platform takes care of such matters itself.

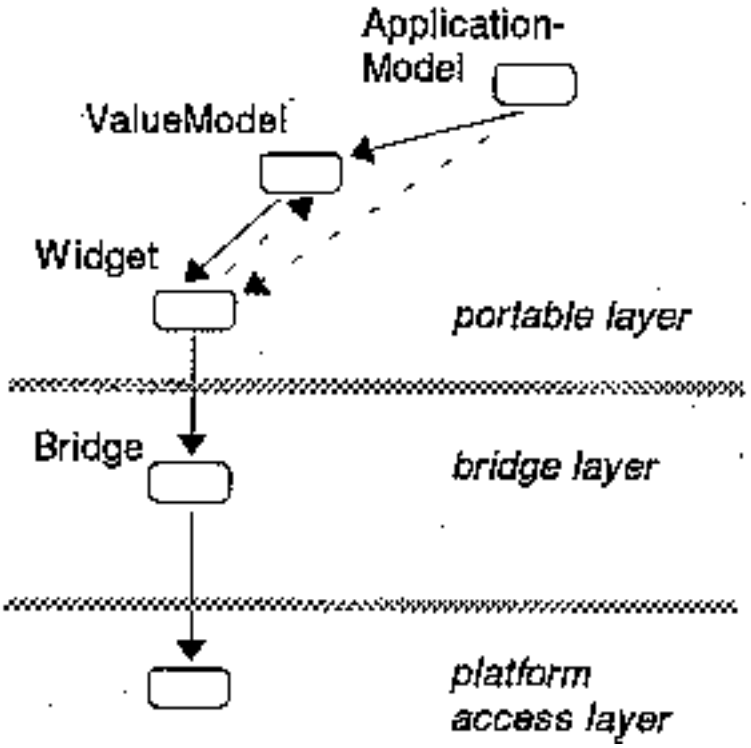
Following this lead, the new architecture arranges to have the Widgets introduced in pattern 2 subsume the responsibilities of both the Views and Controllers in the original framework. Specifically, they are responsible for displaying *and* manipulating the information in ValueModels, and for providing the programmatic API's supported by the Views and Controllers in the original product. The chief benefit of the *MVC* pattern - separating the provision of data from its visual representation - is retained in the new architecture by carrying over the indirect connection from ValueModels to their respective Widgets.

Pattern 6:

Bridge

Bridge

The bridge layer described in pattern 1 is comprised largely of objects fashioned after the BRIDGE pattern defined in [G+95]. These bridge objects are critical to the provision of portability in the new architecture, since they afford the portable objects in the uppermost layer of the system a platform-independent interface to the host.



A bridge object is responsible for taking the native facilities of a particular platform and using them to implement the set of platform-independent capabilities which a particular type of Widget requires to implement its control in VisualWorks. So for example, a RadioButtonWidget would be furnished a Win32RadioButtonBridge on MS Windows, a MacRadioButtonBridge on the Macintosh and so on. Likewise, a ListWidget is matched with a Win32ListBridge, MacListBridge, etc., as appropriate.

The use of bridge objects brings with it a number of advantages:

- It de-couples the inheritance hierarchies of widgets and their implementations. For example, all Win32...Bridge classes inherit MS-Windows-specific capabilities from their abstract superclass Win32Bridge, while Mac...Bridges inherit Macintosh-specific capabilities from MacBridge. Were widgets to be combined with their bridges, the single inheritance model of Smalltalk would render it practically impossible to separate platform-specific capabilities in this fashion, impacting the modularity (and hence the maintainability and amenability to enhancement) of the system very

adversely.

- In VisualWorks, it is possible to alter the characteristics of a control dynamically in such a way that the implementation of that control using a platform's native facilities changes completely. Consider, for example, a VisualWorks push button which is initially assigned a textual label. Implementing such a push-button on Microsoft Windows is relatively straightforward, since the MS Windows button control provides the required capabilities directly. However, VisualWorks also allows an arbitrary DisplayObject (which includes bitmaps, formatted text, geometric figures and graphical composites) to be used as push-button labels. Implementing this kind of button on Windows is significantly more complicated than is the case with the simple textual label. Furthermore, it is possible to change the label of a push-button after it has become visible on the screen. This means that a significant revision of the widget's implementation may be required at any point in its lifetime. Fortunately, this is easy using bridge objects: If a bridge detects a change (a new label, for example) in the widget it's implementing such that it is no longer able to supply the proper implementation, then it simply relinquishes responsibility to a new bridge capable of dealing with the new characteristics.

- The footprint of VisualWorks applications destined to be executed on one platform only can be reduced simply by discarding (or neglecting to load) the code for bridge classes particular to platforms other than the target.

We should note in passing that widgets are actually provided with the appropriate bridges using another pattern - the ABSTRACT FACTORY. In this case, the pattern takes the form of a WidgetPolicy object which returns new bridges in response to messages such as "pushButtonBridge", "listBridge " and the like. Different subclasses of the abstract superclass WidgetPolicy provide suitable implementations of these methods for different platforms.

Pattern 7:

Proxy

Bridge objects actually implement widgets using objects from the platform access layer described in pattern 1. Recall that the purpose of the platform access layer was two-fold: To provide support for the portable objects and to furnish convenient direct access to the platforms facilities for discretionary use by the application developer. To this end (and also to ease the development and maintenance of the bridge objects), we elected to make the platform access layer a little more sophisticated than a mere foreign function interface to the host's API's.

Electing to use the PROXY pattern was a fairly direct consequence of this decision, and is in line with fairly widespread practice, as exemplified by MFC [M94a] and MacApp [A89]. These frameworks employ proxies to encapsulate platform resources such as windows, fonts, bitmaps and the like. Messages to these objects are transliterated fairly directly into API calls involving the encapsulated resources. The proxies may also provide supplementary functionality to make programming more convenient than using the platform API alone. We followed a similar approach

with the platform access layer, producing Smalltalk proxies for windows, menus, bitmaps, fonts, etc. These proxies facilitate most of the manipulations which the hosts' API's support on the corresponding resources. They also provide for a degree of type translation to and from Smalltalk, and for automatic resource reclamation upon garbage collection of the proxy. The proxies for windows and controls are also chief participants in the SASE pattern, discussed in the next section.

Pattern 8:

Self-Addressed Stamped Envelope

Introduced formally in [B95b], the SELF-ADDRESSED STAMPED ENVELOPE (SASE) pattern is frequently employed to provide a means of indirect communication between the lower tiers of a layered architecture. According to this pattern, a nether object may be informed that upon the occurrence of a particular event (the clicking of a button, for example), a specified message is to be dispatched to a specified receiver. Both message and receiver may be varied arbitrarily, and provision is made for the supply of arguments to the message either at the time of registration or message dispatch.

The platform access layers of the experimental system make use of this pattern to allow the interception of host-specific events by objects in higher layers. On Windows, for example, evaluating the expression:

```
winProxy onEventNamed: #WM_PAINT
     send: #paint to: handler
```

causes the (Smalltalk) message #paint to be sent to the object handler whenever Windows dispatches the (Windows) message WM_PAINT to the window whose proxy object is winProxy. Usually, the handler is a bridge object, or an object in the application used to monitor the reception of the Windows message or modify the window's response to it.

Use of the SASE pattern does away with the necessity for direct communication from objects in the platform layer to those in the bridge layer. Eliminating the need to make bridge objects privileged in this way, it makes the full generality of the proxy's capabilities available to the application developer, thus helping to fulfill the requirement (R2, section) for ready access to platform facilities from applications.

The SASE also allows the same window proxy objects to be shared by both bridges and application objects, allowing application developers to modify programmatically the behaviour of interfaces constructed using the VisualWorks screen painter.

Finally, the pattern provides a convenient means of customizing the behaviour of windows without resorting to subclassing. This increases the encapsulation of the platform object classes, leading to easier maintenance and framework enhancement.

Pattern 9:

Flyweight

The SASE pattern requires that each window proxy retain sufficient information that it can dispatch

The SASE pattern requires that each window proxy retain sufficient information that it can dispatch the appropriate message (with arguments, if required) to the appropriate receiver upon the occurrence of a specified event. To take the platform access layer for Microsoft Windows as an example: a naive implementation of the pattern would have each proxy associated with a collection of *registration records*:

$\langle uMsg, msg, rcvr, arg_1, arg_2, \ldots, arg_n \rangle$

Such a record would indicate that upon reception of a Windows message *uMsg*, the Smalltalk message *msg* is to be dispatched to the object *rcvr* with the arguments specified.

Unfortunately, the bridge objects make heavy use of the SASE mechanism, and such an implementation would result in immoderate storage requirements, since each control present in an application's user interface would require several such records to coordinate its activities with the bridges. Furthermore, the need to construct such collections for each control every time a window in the interface is opened would lead to an unacceptable degradation of performance.

These difficulties are ameliorated in the experimental architecture by application of yet another design pattern - the FLYWEIGHT. According to this pattern, economies result from identifying the variable components ([G+95] calls this "extrinsic state") in a large population of similar objects, separating out those components and sharing the identical portions which remain.

The key to using the FLYWEIGHT pattern in the context of the experimental architecture is to observe that bridge objects of a given type use similar registration records. For example, an MS Windows bridge object br1 controlling a top-level window would register the following with its proxy:

```
<WM_SIZE, #size, br1>
  "On reception of WM_SIZE, send #size to br1"
<WM_MOVE, #move, br1>
  "On reception of WM_MOVE, send #move to br1"
<WM_CLOSE, #close, br1>
  "On reception of WM_CLOSE, send #close to br1"
```

Likewise, top-level window bridge br2 would register with its proxy:

```
<WM_SIZE, #size, br2>
  "On reception of WM_SIZE, send #size to br2"
<WM_MOVE, #move, br2>
  "On reception of WM_MOVE, send #move to br2"
<WM_CLOSE, #close, br2>
  "On reception of WM_CLOSE, send #close to br2"
```

Clearly, the registration records for both bridges differ only in the receiver they stipulate. We can remove the receiver field from the registration records to produce a single collection of abbreviated records we'll name rcds:

```
<WM_SIZE, #size>
  "On reception of WM_SIZE, send #size"
```

```
<WM_MOVE, #move>
  "On reception of WM_MOVE, send #move"
<WM_CLOSE, #close>
  "On reception of WM_CLOSE, send #close"
```

Now bridge br1's registration records can be synthesized directly from the pair <br1, rcds>, and br2's records from <br2, rcds>. Note that the collection of abbreviated records is shared by both pairs. Using such pairs to register bridges with their proxies substantially reduces the storage overhead associated with the SASE implementation in the system. Furthermore, the shared components of such pairs (rcds in the example) can be pre-computed at system start-up, and this speeds up the process of interface construction significantly.

Pattern 10:

Accumulator

As far as we're currently aware, this is the only truly novel design pattern conceived during the course of the experiment. Its discovery and use arose during the implementation of the platform access layers. To take Microsoft Windows as an example once again, consider a Win32 API[M93] call which creates a new dialog window of window-class "MY_DIALOG_CLASS":

```
CreateWindowEx(
  WS_EX_DLGMODALFRAME,
  "MY_DIALOG_CLASS",
  "A Dialog",
  WS_VISIBLE | WS_CAPTION | WS_SYSMENU |
     WS_MINIMIZEBOX | WS_MAXIMIZEBOX,
  CW_USEDEFAULT, CW_USEDEFAULT,
  CW_USEDEFAULT, CW_USEDEFAULT,
  NULL, NULL, hInst, NULL)
```

It's easy to appreciate the inconvenience that would be associated with using a direct transcription of such a call in Smalltalk. The experimental system seeks to address this by providing objects called *creators.* The function of such an object is to provide the arguments to a CreateWindowEx call. These arguments may be provided as defaults, or they may be set individually by the creator's client. To create the dialog window in the example above, the developer would begin by procuring an appropriate creator object - usually by requesting one from the object representing the window-class in question. The creator comes loaded with a number of default parameter values, requiring that the user supply only those for which the default values are inappropriate. Using such a creator, the call above can be rendered much more succinctly:

```
myDialogClass creator
   windowName: 'A Dialog Window';
     "Creator forms disjunction of the following..."
   visible; caption; sysMenu; minimizeBox; maximizeBox;
```

```
createWindow.
```

The deployment of creator objects is a particular example of the ACCUMULATOR pattern. The idea behind this pattern is to encapsulate in a single object the parameters required to control one or more complex operations. Examples abound of its use, especially in graphics systems. They include the DeviceContexts of the Windows GDI [M93], GraphicsContexts in X-Windows [J88], or - to cite a more strictly object-oriented example - the BitBlt (later RasterOp) objects of Smalltalk-80 and its descendants [GR83].

The ACCUMULATOR pattern confers a number of advantages in such applications:

- Accumulators may be pre-loaded with default parameter values, relieving the application programmer of the need to specify each parameter explicitly for every operation. Accumulators may be produced with a variety of different pre-loaded values by making the appropriate requests to other objects in the system. In the example above, the object representing the MS Windows window-class produces a creator object with the appropriate window-class name and style already set.

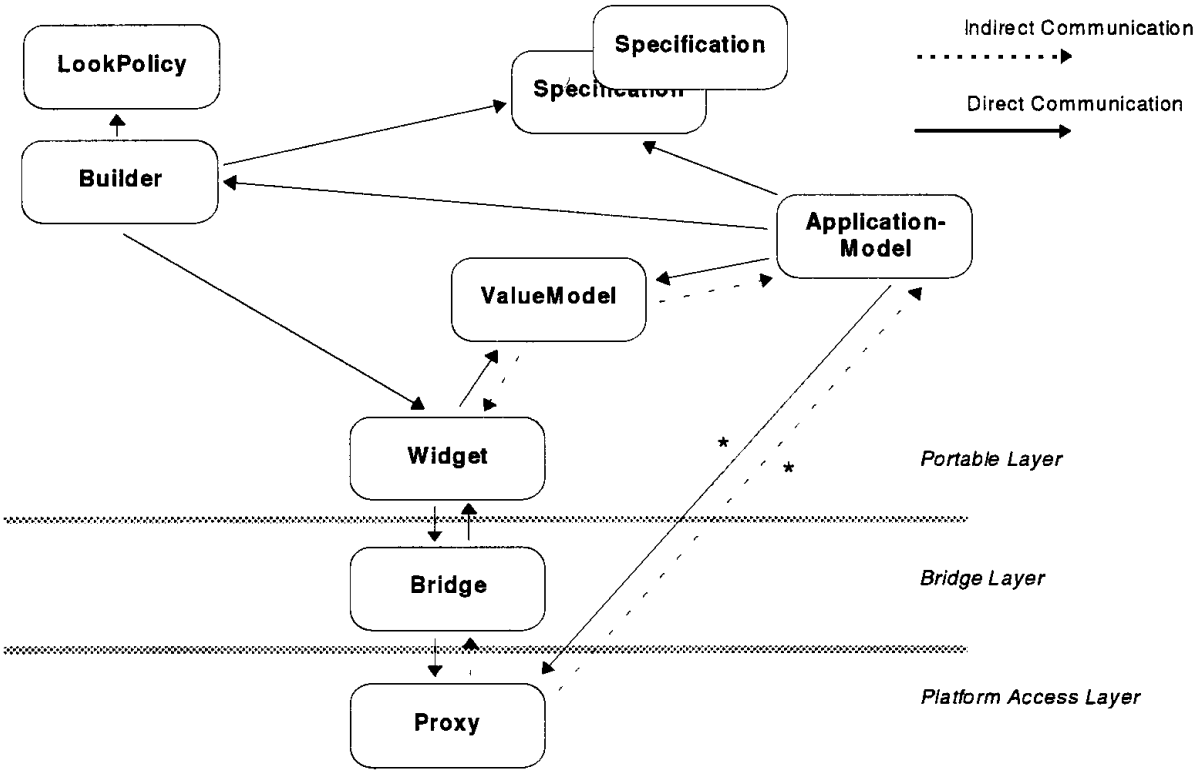


Figure 2. The Experimental User Interface Architecture

- Accumulators can provide extra facilities for setting parameters. An example of this can been seen in the use of the creator object described above. In the C call, the style of the new window is specified as a complex disjunction of masks. By contrast, the creator object forms the disjunction itself, requiring only that the programmer send the appropriate unary messages.

- If a succession of operations require parameters which are substantially similar, the same accumulator can be reused for all operations. This provides greater convenience, and - as shown by the X-Windows system - a potential for added efficiency.

5. The New VisualWorks Architecture

Figure 2 is a schematic of the architecture which resulted from the experiment. Objects used only on a transitory basis (such as creators), or those largely invisible to the application programmer (for example, registration records supporting SASE communication) have been omitted. The communications annotated with asterisks are optional, and only arise if the application programmer elects to make direct (non-portable) use of elements of the platform access layer.

6. Conclusion

Incorporation of design patterns into the experimental architecture enabled the project successfully to fulfill all of the requirements set out for it. We were able to reconcile a high degree of portability with the use of and ready access to host facilities, to meet our goals for backwards compatibility and to provide the performance and maintainability required.

We can draw several broad conclusions from the experiment concerning the use of design patterns, particularly in the context of framework evolution of the kind detailed here:

- Successful use of design patterns to guide (rather than simply document) architectural evolution in a real-world, commercial application is indeed possible. Of the ten patterns discussed in the above, only one (ACCUMULATOR) is actually new.

- Given the current state of the art, it is not possible simply to apply patterns mechanically during the design process. In almost every instance, we found it necessary to adapt or tailor existing design patterns so as to fashion them for use in the architecture. To make such adaptations successfully requires broad-based architectural experience, and recognition of the trade-offs which can be made in each particular instance.

- The application of a pattern does not invariably improve an architecture. In some cases (examples in this case are the DECORATOR and MODEL-VIEW-CONTROLLER patterns), it may be advisable during evolution to discard or substantially modify a pattern which has become inappropriate due to changes in requirements.

- As was been observed in [G+95], design patterns rarely stand in isolation. Often, one or more design patterns participate in the implementation of another: Examples in the new VisualWorks architecture are BUILDER and ABSTRACT FACTORY (components of an INTERPRETER); Adapter, BRIDGE and PROXY (LAYERED ARCHITECTURE); FLYWEIGHT (SASE). Thus it its very common to find a single object participating in a number of different design patterns.

- In common with a other authors, experience with the evolution of the VisualWorks architecture suggests that as an architecture is refined, it is common to find that increasingly substantial portions of the implementations of an object's responsibilities are delegated to other objects. In [JO93], this is manifest as the replacement of inheritance with composition and delegation; in our case, we can observe it in the delegation of I/O responsibilities from Widgets (erstwhile Views and Controllers) to Bridges and (indirectly) Proxies.

Acknowledgements

The author owes a considerable debt of gratitude to the members of the erstwhile *van Gogh* project team at ParcPlace-Digitalk (Terry Chou, Mike Khaw, Lee Ann Rucker, Lisa Yee-Estrada , together with a number of part-time contributors), who collaborated in the work documented in this report.

References

[A87] Apple Computer Inc. Human Interface Guidelines: The Apple Desktop Interface. Addison-Wesley, 1987.

[A89] Apple Computer Inc. *MacApp 2.0 Programmer's Guide.* 1989.

[B95a] Brown, K. Remembrance of things past: layered architectures for Smalltalk applications. *The Smalltalk Report* 4(9):4-7, 1995.

[B95b] Brown, K. Understanding inter-layer communication with the SASE pattern. *The Smalltalk Report* 5(3):4-8, 1995

[C93] Custer, H. *Inside Windows NT*. Microsoft Press, 1993.

[C95a] Coplien, J. A generative development-process pattern language. in [CS95].

[C95b] Cluts, N. *Programming the Windows® 95 User Interface.* Microsoft Press, 1995.

[CS95] Coplien, J., Schmidt, D. *Patterns Languages of Program Design.* Addison-Wesley, 1995.

[DeB95] DeBruler, D. A generative pattern language for distributed processing. in [CS95].

[G+95] Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GR83] Goldberg, A., Robson, D. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley,

1983.

[H92] Haggar, P. New controls in OS/2 2.0: an overview. *IBM Personal Systems Developer*, No. 1, 1992.

[JO93] Johnson, R., Opdyke, W. Refactoring and aggregation in object technology for advanced software. Nishio S., Yonezawa, A. (eds.), *LNCS 742*, Springer, 1993.

[J88] Jones, O. *Introduction to the X Window System.* Prentice Hall. 1988.

[KP88] Krasner, G. E., and Pope, S. T. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3), 1988.

[L+95] Lewis, E., Rosenstein, L., Pree, W., Weinand, A., Gamma, E., Calder, P., Andert, G., Vlissides, J., Schmucker, K. *Object Oriented Application Frameworks.* Manning Publications, 1995.

[M91] Microsoft Corporation. *The Windows Interface: An Application Design Guide*. Microsoft Press, 1991.

[M93] Microsoft Corporation. *Win32 Programmer's Reference.* Microsoft Press. 1993.

[M94a] Microsoft Corporation. *Manuals for Visual C++ and Microsoft Foundation Class Library.* 1994.

[M94b] Microsoft Corporation. *OLE Programmer's Reference.* Microsoft Press, 1994.

[P72] Parnas, D. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), 1972.

[P94] ParcPlace Inc. *VisualWorks Tutorial.* 1994.

[P96] ParcPlace-Digitalk Inc. *VisualWave: Building Live WWW Applications.* http://www.parcplace.com/marketing/products/vwave/wpaper.html.

[S95] Schmid, H. Creating the architecture of a manufacturing framework by design patterns. *Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, 1995.

[T88] Tanenbaum, A. *Computer Networks (2nd ed.).* Prentice Hall, 1988.