An Architecture For Evolving Graphics Applications

Michael L. Davis

Software Farm, Inc.
(michael@bhi.com)

August 10, 1992

Abstract

An important aspect of the architecture of interactive systems is the isolation of the functional aspects of the system from each other as well as from the user interface. This isolation often consists only of the separation of the programming code modules, ignoring as too complicated the further goal of separation of knowledge as well.

By the separation of knowledge it is meant that the application knows absolutely nothing about how it is represented visually, and conversely the user interface knows absolutely nothing about what it is representing.

This paper presents a scalable architecture to support this separation and discuses examples of the use of this architecture in various application domains.

Keywords - User Interface, Application Framework, User Interface Management System

1.0 Introduction

An evolvable architecture for any type of application has a number of notable qualities. These qualities consist of minimizing programming effort and software defects under the following stressful conditions:

- Changes to the functionality of the application
- Changes to the type of software technology used to provide the application functionality
- Changes in the software personnel who are responsible for a the application software
- Changes to the number of hardware platforms supported.
- Changes in the underlying software standards not envisioned when the application software was written

It is proposed that these qualities are present in architectures that have the following attributes:

1. Provides a paradigm of the overall program structure, indicating where each major component of functionality should be placed within the architecture and how the components should interact with each other.
2. Provides the guidelines to enable, with minimal cognitive load, straight forward and timely implementation.

3. Allows for future changes to any component to be as straight forward and to have as little impact on the other components as possible.
4. Supports the reuse of major components in other similar applications.
5. Easily supports the incorporation of new or unanticipated functionality into existing applications.

It should also be straight forward to design, implement, debug, modify and maintain. Applications written with an architecture with these qualities will result in these same qualities being incorporated into the design and implementation of the internal components of the architecture framework.

2.0 Nomenclature

The terminology of software system architectures is still non-standardized and so the nomenclature used in this paper will be briefly defined. An *application* is a group of software components collected together into a unified whole. A *component* is a subset of an application that supplies a single service. The *semantic* component is the part of an application that contains the software to related to the purpose of the application. The *presentation* component is that part of the application that supports the user in viewing and interacting with the application. *A Functional* components is a semantic or presentation component. The VFACE component is the remaining part of the application that ties the other parts together (i.e the brains and glue) and makes the software a whole application instead of just a group of components

3.0 Overview

We call this architecture the ‘VFACE’ architecture.

The VFACE name is derived from the letter V (the top two vertices, representing the Presentation and Functional components, are only connected by a third vertex, representing the VFACE component) concatenated with the word interFACE.

This architecture requires that the Functionality component and Presentation component be completely separated and made completely dependant on (slaves of) a third, master component, the VFACE. The VFACE component contains the operational control (i.e. the brains, the glue) of a program. In other words, the same Functionality component and Presentation component can be combined using different VFACE components to create, operationally, quite different applications.

This architecture can be thought of as a macro-object-oriented design or a miniature client-server architecture

Elements of a well designed VFACE architecture are:

- That the functional components have no knowledge of each other
- That the functional components are completely dependant on the VFACE component. They respond to commands and requests and may only send data and information about changes to the VFACE that were requested
- The knowledge the VFACE component has about the functional components is minimal

3.1 Command Flow

The control for an application using the VFACE architecture is contained entirely in the VFACE component. All other components act as servers and only supply services to the VFACE component (see Figure 1 on page 3).

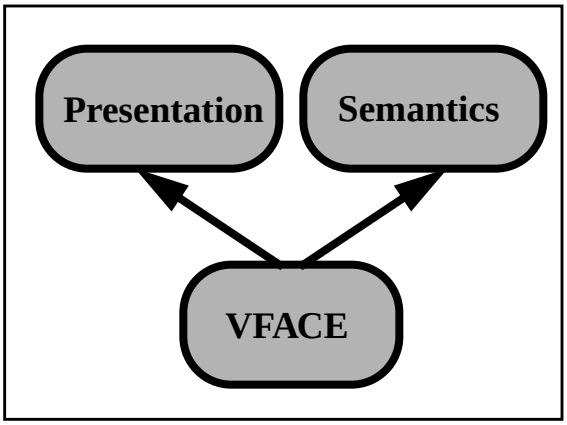

FIGURE 1. Command flow in the VFACE Architecture.

3.2 Information Flow

Information flows between the VFACE component and the other components only. Information may not flow directly between two functional components (see Figure 2 on page 3). When information is sent to the VFACE component, it is only in response to a previous request by the VFACE component that it do so (possibly during initialization).

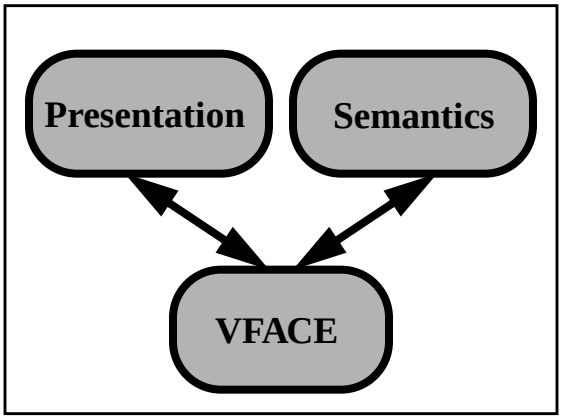

FIGURE 2. Information flow in the VFACE Architecture.

3.3 Segmentation of the Major Components

The architecture can be expanded to include other specialized functional components (see Figure 1 on page 4). The presentation and semantic components can be decomposed into other, independent sub-components. This decomposition can proceed to any depth using the design technique called *vertical partitioning*.

4.0 VFACE Design and Implementation Methodologies

4.1 Design

The design of an application using the VFACE architecture consists of assessing the optimal composition and style of the functional components, This can be reduced to the following steps:

- Determine what the functional components will be
- Determine what public interface each functional component will present to the VFACE component
- Determine what the operations are that the VFACE component will provide
- Determine what requirements the VFACE component demands of the public interface to the functional components (with respect to the operations the VFACE component will preform using these interfaces).

4.2 Implementation

The implementation of an application using the VFACE architecture is divided into implementing the functional components and the VFACE controller. The functional components are implemented as libraries.

As for event driven functional component designs (for example user interfaces) each component is given the name of a destination, in the VFACE component, to which it will send messages with specific data which the VFACE component will process and react as appropriate.

The specific implementation of the VFACE component itself is not specified by this architecture (however see “Experience” on page 5). Very economical implementations are possible for the development of prototypes and complex implementations are possible for more complex applications.

All linkage and program state machine code goes into the VFACE component. As a rule-of-thumb, all code that does not seem to fit into any component gets put into the VFACE component by default.

5.0 Advantages and Disadvantages

5.1 Advantages

Separating the components allows any component to be redesigned, rewritten, ported to another platform

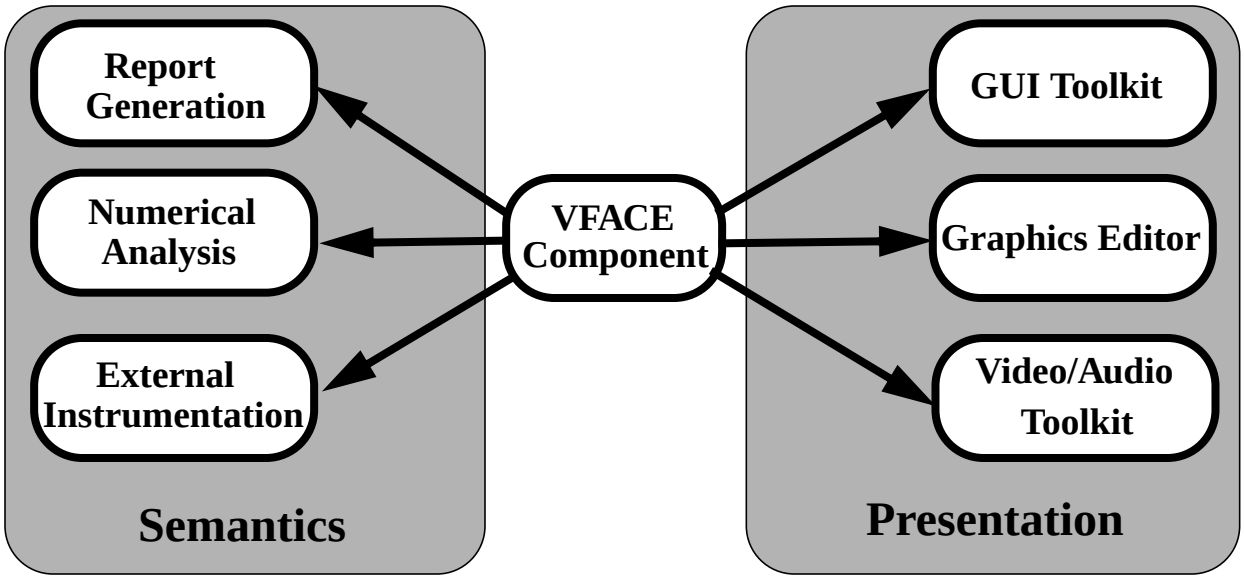

FIGURE 1. Adding Components to the VFACE Architecture

or be run over a network without having to change anything in any other components.

Functional components are conceptually like old fashioned libraries (even though they may be implemented in an object-oriented fashion). They contain no specific program semantics or behavior, just functionality. This makes the functional components reusable in other applications that require the same or similar functionality, as proven with other libraries like the standard 'C' library. This also makes these components easy to write as their design is already familiar to many programmers.

5.2 Disadvantages

Designing functional components is isomorphic to designing objects. And similar to the task of object-oriented programming in-the-small, there is a requirement for a good deal of design time up front, deciding what the components will be and, more difficult, what their public interfaces will be.

There has been some arguments about the ability to have a clean design (code separation) in the face of performance requirements, especially when one of the functional components consumes vast quantities of time (for example a 3D graphics component). The overhead, with respect to execution time, of the VFACE architecture is insignificant because of the macro nature of the functional components. This can be qualitatively verified by examining the ratio of the amount of time spent by an executing program spent within the code specifically added to separate the functional components to the time spent within the sluggish functional component.

6.0 Related Work

The generality of the VFACE architecture encompasses many previous designs with sometimes subtle, but essential alterations.

Most applications today make calls directly to a graphics library standard. This provides no standard isolation at all between the semantic component of the application and its presentation. A better approach is for the application to make calls to a layer of code 'wrapped around' the graphics library. This provides some isolation of the application from any future changes made to the graphics standard library as well as providing portability to other graphics libraries.

The Model-View-Controller (MVC) design [Krasner and Pope, 1988], is quite good except that it permits the View (the presentation component) to access the Model (the semantic component) directly.

The Seeheim architecture [Plaff, 85] is similar to the MVC architecture except that also permits unneces-

sary communication between components, namely the semantic component is allowed to access the presentation component directly.

The Abstraction-Link-View design [Hill, 1992] uses a sophisticated constraint manager to manage the links between Abstraction (semantics) and View (presentation). Unfortunately the control of the application remains in the semantic component. To alter this design to gain the benefits of a VFACE architecture one could use a Abstraction-Link-VFACE-Link-View approach (where control would be moved to the VFACE component).

7.0 Experience

The following sections describe our experience with developing applications using the VFACE architecture.

7.1 VFACE Controller Implementation

Examples of internal VFACE designs and methods of implementation are:

A collection of a large number of individually coded routines to interpret user interface events, determine specific responses based on the current application state, and send corresponding commands to the presentation and semantic components. We call this the *brute-force* method.

A system of high-level event handlers [similar to Green, 86] which are networked together and are programmed to handle user interface (presentation) events and semantic events.

A system where there are many small data objects that are shared across partition boundaries, each containing semantic data (accessible only by the semantic and VFACE components), presentation data (accessible only by the presentation and VFACE components) and data needed by the VFACE component itself.

7.2 Program Examples

Examples of applications built using the VFACE architecture:

7.3 A Direct-Manipulation Editor for Displaying and Editing Diagrams

In this application the VFACE component was written using the *brute-force* method (see “VFACE Controller Implementation” on page 5). The relative size of the VFACE component is about 6% of the size of the entire application (in lines of code). Minimizing the size of the VFACE component is felt to be a good thing. Minimizing the knowledge that the VFACE component has of the specifics of the semantic component and presentation component is also felt to be a good thing. This was a straight forward, simple implementation task.

7.4 A VFACE architecture development environment

In applications generated using this development environment, the VFACE component consists of an interpreter driven by a high-level object-oriented language [VisualADE, 92]. The language describes the ‘glue’ and constraints between the various functional components. In addition, a large number of presentation components are provided to ease development of direct-manipulation editors and standard user interfaces. These components conform to a standardized interface that the VFACE component interpreter recognizes.

8.0 Conclusion and Future Work

The VFACE architecture provides a useful implementation paradigm for almost all programs including those containing graphics, databases, multimedia and external processes.

We are currently doing research to determine the best method of adding a reusable persistence scheme to the architecture. A particularly likely candidate is the treatment of the VFACE component as a

persistent shared memory controller (see “VFACE Controller Implementation” on page 5).

We are also currently researching methods of adding rules to the system. We are especially interested in supporting design rule checking (both heuristic rules which suggest operations and strict rules which generate and validate operations).

9.0 References

Krasner and Pope (1988). A cookbook for using the model-view-controller user interface paradigm in smalltalk-80, *Journal of Object-Oriented Programming* August/September, pp. 26-49.

Plaff, G. E. (ed.) (1985). *User Interface Management Systems*, Heidelberg: Springer-Verlag.

Green. Mark (1986). A Survey of Three Dialog Models, *ACM Transactions on Graphics, Vol. 5, No. 3,* pp. 24-275.

VisualADE (1992). *User’s Guide and Programmer’s Manual,* Software Farm, Inc.

Hill, Ralph D. (1992). The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications, *ACM CHI Conference Proceedings*, pp 335-342.