Software Architecture: Foundation of a Software Component Marketplace

E. James Whitehead, Jr. Jason E. Robbins Nenad Medvidovic Richard N. Taylor

Department of Information and Computer Science
University of California, Irvine
Irvine, California 92717-3425
{ejw,jrobbins,neno,taylor}@ics.uci.edu

Abstract

This paper provides a characterization of software architectures which become the foundation for the establishment of marketplaces for software components. A description is made of a framework of key properties that a software architecture should exhibit to be the basis of a component marketplace. As examples of existing software component marketplaces, Unix filter and Visual Basic VBX marketplaces are examined with respect to this framework. The Chiron-2 architectural style is analyzed with respect to these properties to determine its potential as a future software component marketplace in the user interface domain.[1].

1.0 Introduction

The goal of a software components marketplace, though decades old, is still compelling. In a component based development scheme, instead of hand-crafting systems, software designers browse catalogs of software components from multiple vendors, assembling complex systems from abstract building blocks. Many benefits result from this scheme, including decreased system development time and cost, low component cost due to the amortization of component development costs over multiple users, robust components due to greater maintenance resources supported by multiple users of each component, and a wide range of general-purpose and domain specific components. The benefits of component based software development have motivated many researchers, from its introduction by McIlroy in 1968 [McIl69] to the present day work on megaprogramming [BoSc92] [Trac91], domain specific architectures [Haye94] [ADAG92], hierarchical systems with reusable components [BaOM92], and formal modeling of components and architectures [AAG93].

Existing marketplaces, such as those for digital electronic components, share the common feature that they are based on the existence of a common architectural standard. Digital electronics components interoperate based on a simple architecture, usually based on the mapping of +5 volts to a logical 1, 0 volts to a logical 0, with signals passed through wires. More complex components build upon this standard, for example adding clocking and bus conventions in the case of the architecture for microprocessor component families. In all cases these simple standards are supplemented by rules detailing meaningful connections and configurations of components.

A commercial software component marketplace is the coming together of buyers and sellers to exchange money for software components. The relationship of an architecture to a marketplace is that an architecture is a necessary but not sufficient condition for the creation of a marketplace. Just as an electronic components marketplace could not exist without the existence of the digital electronics architecture, software component marketplaces cannot exist without appropriate architectural standards. Architectures provide a foundation upon which component marketplaces are potentially built.

Non-commercial software component marketplaces are also possible. In these marketplaces, components are given away without direct monetary compensation in the expectation of achieving some other goal. Sometimes the motivation is purely altruistic, in the tradition of the Internet

1. This material is based upon work sponsored by the Air Force Materiel Command, Rome Laboratory and the Advanced Research Projects Agency under contract number F30602-94-C-0218. The content of the information does not necessarily reflect the position or policy of the Government and no official endorsement should be inferred.

community. With other exchanges, there is an expectation of increased prestige or recognition derived from the positive experiences of community members using the software component. Just as for the commercial case, software architectures provide a potential basis upon which non-commercial software component marketplaces may be built. In fact, a well-designed architecture should support both types of component marketplaces, commercial and non-commercial.

In the remainder of this paper, we describe proposed requirements a software architecture must meet to be the basis of a component marketplace. Note that these requirements have not been entirely validated by existing research, and should be considered a framework for further research into what should be the complete set of requirements. These requirements are used to examine a software architecture introduced decades ago, the Unix pipe and filter architecture, and to examine an architecture introduced more recently, the Visual Basic VBX architecture. Finally, a current research architecture, Chiron-2 [TMA+95], is analyzed with respect to these properties to determine its potential as a future software component marketplace in the UI domain.

2.0 Requirements on Architectures for Software Component Marketplaces

From an economic standpoint, architectures have been defined as "a complex of standards and rules" [MoFe93], while from a software engineering standpoint architectural styles are essentially key design idioms. That is, the standardizing aspects of architectures carry greater relevance to an economic frame of reference, while structures for the design of systems are more relevant to the software engineer. These two viewpoints show that aspects of a given architecture are valued differently, depending on the factors most relevant to the observer.

While the motivation for examining the key aspects of an architecture that make it a good foundation for a component marketplace is primarily economic, the requirements listed below are technical in nature. The strength of this approach derives from choosing a economically-motivated goal (successful component marketplaces), and emphasizing technical requirements that help reach that goal. By choosing an economic goal, the resulting requirements are in some cases quite different to those embodied by current software architectures.

There are a myriad of factors beyond these architectural requirements which affect the commercial success of a component marketplace, such as amount of capitalization, support by key players, backing by smaller companies willing to develop components, timing, marketing strategy, access to distribution channels, and pricing. However, we feel that meeting these requirements can substantially enhance the chances a given architecture can become a successful marketplace.

The key properties of an architecture which make it suitable as a potential basis for a component marketplace include:

1. Multiple component granularities

   The architecture must support components that are both small and large, for example, as small as a linked list and as large as a spelling checker. While most larger components could undoubtedly be constructed from smaller components, larger components can provide a more meaningful packaging of functionality to designers. In most cases, a buyer who wants a spelling checker component would find a pre-made spelling checker component more relevant than more generic components which can be used to construct the spelling checker.

2. Substitutability of components

   The architecture must provide support for removing one component and substituting for it an equivalent component. This allows competition based on features and price for a particular component. For example, a word processor vendor might want to replace their current spelling checker component with one that is cheaper, or more tailored for a niche market which has a highly specialized vocabulary. As another example, in the user interface domain it might be desirable to replace one low-level graphical toolkit with another, perhaps less expensive toolkit.

3. Parameterizable components

   The architecture must support components which can be parameterized before being used in a design, allowing the behavior of the component to be tailored by the designer. For example, a sorting component might be parameterizable based on characteristics of the input set, and different designs would use the same component with differing parameterizations. Ideally, the parameterization of a component should be easy to perform, aiding the ease of use of the component by the designer. When available and appropriate, a GUI interface is the preferred interface to the component for performing this parameterization. Note that this interface does not necessarily have to be supplied by the component itself, but could instead be a scaffolding that is used during design, then removed for execution.

4. Customizable components

   To support customization by the end-user, the behavior of the component must be customizable via a facility provided by the component. The customization interface should be tailored to the skill level of the user, and should have an emphasis on its usability.

5. Component development in multiple programming languages

   Since different programming languages have strengths for different applications, and since new major languages emerge periodically, an architecture should sup-

port components developed in different programming languages. Additionally, an architecture should provide support for legacy systems, which will likely be written in multiple programming languages.

6. Component-specific help

   Ideally, it should be possible for software designers and component users to get help from the component itself. This might vary from an embedded phone number for a support line to a help window or an automatic connection to a WWW server which contains documentation and a frequently asked questions list.

7. Composing component-specific user interface dialog and presentation properties

   While there are many components which do not present an interface to the user, there are some domains and some components which do have stereotypical user interfaces. For example, a component which provides modem dialing functions may present an interface of phone numbers, while a spreadsheet artist component may present a grid view of the spreadsheet data. In the case where there are multiple component-specific user interface dialog and presentation properties, an architecture must support their composition into interfaces which can be tailored to meet the specific requirements for the application user interface. For example, a program which automatically dials a modem and downloads data for display and manipulation by a spreadsheet component should look like it has an integrated interface tailored to its specific task and typical users rather than something haphazardly patched together.

8. Easy distribution of components from seller to buyer

   It should be easy to package and distribute a component. Ideally, an architecture should support the distribution of components in binary form, so they can be used without need for a compilation step. This supports ease of use: a component can be copied off of a CD-ROM, or received from a network in a ready-to-use format.

9. Support for multiple sales models

   Existing software has multiple sales models, ranging from single sale, single-user (PC software model), to single sale, multiple users (workstation model). In a highly vertical market it is common to give away evaluation copies that expire after a given time period. An architecture should not be biased towards only one sales model. Also, an architecture should ideally provide support for differing sales models, for example by providing a standard license handling component for the workstation sales model.

Fundamental to these requirements is the notion that components should be designed with an emphasis on usability. Unlike traditional usability of a product interface where the only significant interface is between the program and the end-user, components potentially have two interfaces: one between the component and the designer, and another between the component and the end-user. Like traditional applications, an emphasis on usability in components is expected to provide benefits in ease of use, flexibility, lower learning time, and ease of adoption through a reduction of interface complexity.

3.0 Existing Software Component Marketplaces

We will now examine two architectures which have both achieved significant success. The first, the Unix pipe and filter architecture, achieved significant success in forming the basis of a primarily non-commercial components marketplace comprised of many filters accessible via the Internet. A more recent example is the Visual Basic VBX architecture, which has formed the basis of a successful commercial components marketplace. These two architectures will be described below, followed by an analysis of how they each satisfy the requirements of Section 2.

3.1 Unix Pipe and Filter

The Unix pipe and filter mechanism, provided by all major shell programming languages for the Unix operating system, is one of the oldest component based architectures. Using pipes, the output from one component becomes the input to another component. A component typically acts as either a source of data, a sink for data, or a transformational filter which converts input data into a different form of output data. Components interact by passing streams of untyped data. Using the language facilities of the shell, components can be combined in very intricate ways. As well, languages like *awk* and *sed* allow for the creation of filters with complex behaviors.

The power and flexibility of this architecture has influenced many other operating systems, and has created a sizeable non-commercial components marketplace. Many components are available, ranging from chess games to file format converters to compilers. A smaller commercial components marketplace also exists, exemplified by printer filters that convert different output file formats into the page definition language understood by a particular printer.

Unix pipe and filters meet many of the requirements given in Section 2 (referred to by number in the paragraph below) for becoming the basis of a software components marketplace. Filters can be of arbitrary granularity (#1), and can easily be substituted for one another (#2). Limited numbers of parameterizable components (#3) do exist, such as the programmable filters, *grep*, *sed* and *awk*, but these are more the exception than the rule. While some filters are customizable through initialization files or command line options, they are typically weak in this area (#4). Filters have been written in every programming language imaginable (though mostly in C), easily meeting the requirement for independence of programming language (#5). Most filters do offer limited component-specific help (#6) from within the filter, typically on command-line options. How-

ever, almost all filters do come with *man* pages describing their operation.

Since Unix filters predate graphical user interfaces, it is unfair to be critical of their lack of composable interface widgets (#7). However, a major flaw of the pipe and filter mechanism is the difficulty of adding graphical interfaces to filters. Filters are typically distributed in source code, a major drawback which effectively limits the acquisition of new filters to a Unix-savvy elite (#8). Unix filters also only support a single sales model of buy once, use by everyone. There is no support for accessing license servers or a pay-per-use mechanism. (#9). It is perhaps the lack of graphical user interface support, the high skill levels needed to utilize existing distribution channels, and the lack of flexibility in sales model which has favored a non-commercial filter marketplace over a commercial marketplace.

3.2 Visual Basic and VBX

Microsoft Visual Basic [Aitk93] provides a graphical development environment for Microsoft Windows integrated with a direct manipulation GUI builder. Graphical user interface elements and their graphical and functional attributes may be specified interactively through property sheets. For example, after adding a list box to the GUI, a developer may specify such attributes as size, position, color, and whether multi-item selections are allowed. Additional GUI elements may be seamlessly integrated into the development environment through the use of Visual Basic Extensions (VBXs). VBXs package the graphical and functional properties of a new user interface element.

Visual Basic components have several advantages over software libraries. It requires very little programming effort to integrate a VBX into a new or existing application since the user interface and the functionality are provided in a single easy-to-use package. Additionally, various attributes of the component, both visual and functional, can be customized by the programmer through the property sheet of a VBX. VBX developers, not the VBX users, are responsible for creating the GUI interface and property sheets. The combination of the GUI builder and the property sheets allow interactive use, configuration, and testing of Visual Basic components in an environment closely resembling that of the application environment.

In contrast to VBXs, software libraries typically provide only functionality and require the programmer to design the user interface. Software libraries also burden the developer by requiring written code to configure and tailor the library for application use. Software libraries are more rigid and time consuming to use for Windows user interface programming than Visual Basic components. Visual Basic components provide functionality with little programming effort, significantly reducing the barrier to learn, use, and reuse software.

The down side to Visual Basic is that it is tightly tied to Windows and its limitations of a segmented memory model, cooperative multitasking, and lack of robustness. Because of this, those developing applications in Visual Basic using VBXs are encountering problems scaling up their applications [Udel94]. The Visual Basic/VBX architecture is simple, providing primarily a common interface for event passing and Windows API calls.

Despite the simplicity of the Visual Basic/VBX architecture, it still meets many of the requirements for a successful architecture. While there are VBX components of varying size (requirement #1 from Section 2, hereafter referred to by number), even components within the same domain (such as databases) are not substitutable for one another (#2). Components are customizable and parameterizable to a limited degree using graphical property sheets as an interface (#3, #4). VBXs may be written in a limited number of languages, notably Visual Basic and C++, providing partial multi-lingual support (#5). While component-specific help is possible, existing components do not provide component-specific help from within the component (#6). Visual Basic provides strong support for composing user interfaces using widgets from different components (#7). Components can be easily distributed on disks from vendor to customer, and can be used as-is, without need for recompilation (#8). VBXs do not support multiple sales models since they only support the single-sale, single-user model typical on PCs (#9).

We now examine a new research architecture, the Chiron-2 architecture by introducing its features, and examining its suitability for providing a potential basis for a component marketplace.

4.0 Chiron-2 Overview[2]

The Chiron-2, or C2, architectural style [TMA+95] is designed to support the particular needs of applications that have a graphical user interface aspect, but the style clearly has the potential for supporting other types of applications. A key motivating factor behind the development of the C2 style is the emerging need, in the user interface world, for a more component-based development economy. User interface software frequently accounts for a very large fraction of application software, yet reuse in the UI domain is typically limited to toolkit (widget) code. The C2 style supports a paradigm in which UI components, such as dialogs, structured graphics models (of various levels of abstraction), and constraint managers, can more readily be reused. A variety of other goals are potentially supported. These goals include the ability to compose systems in which components may be written in different programming languages, components may be running in a distributed, heterogeneous environment without shared address spaces, architectures may be changed dynamically, multiple users may be interacting

2. This section is a summary of [TMA+95] which will appear in the Proceedings of the 17th International Conference on Software Engineering.

with the system, multiple toolkits may be employed, multiple dialogs may be active (and described in different formalisms), multiple media types may be involved, and multiple user tasks ("processes") supported. We have not yet demonstrated that all these goals are achievable or especially supported by this style. However, we have examined several key properties and built several diverse experimental systems, and believe that our preliminary findings are encouraging and that the style has substantial utility "as is."

The C2 style can be informally summarized as a network of concurrent *components* hooked together by *connectors*, i.e., message routing devices. Components and connectors both have a defined top and bottom. The top of a component may only be connected to the bottom of a single connector. The bottom of a component may only be connected to the top of a single connector. There is no bound on the number of components or connectors that may be attached to a single connector. When two connectors are attached to each other, it must be from the bottom of one to the top of the other (see Figure 1).

Each component has a top and bottom domain. The top domain specifies the set of *notifications* to which a component responds, and the set of *requests* that the component emits up an architecture. The bottom domain specifies the set of notifications that this component emits down an architecture and the set of requests to which it responds.

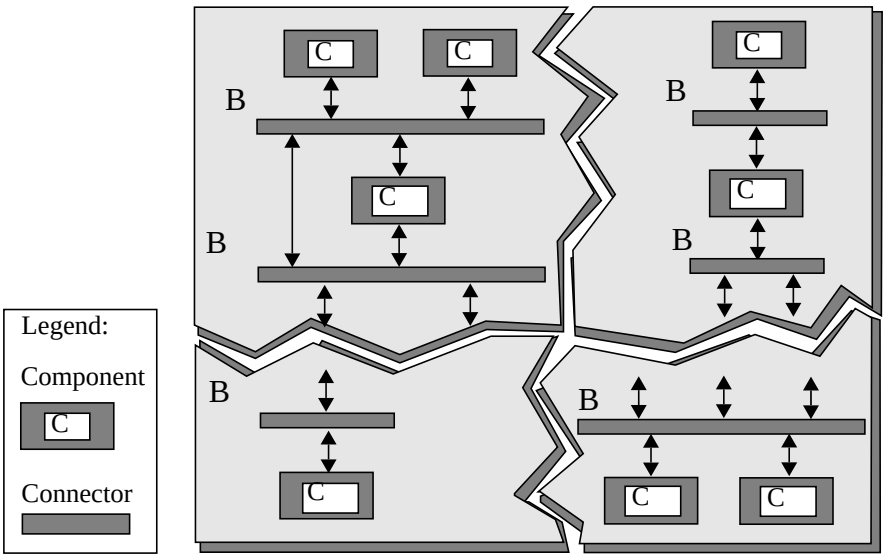
All communication between components is solely achieved by exchanging messages. This requirement is suggested by the asynchronous nature of component-based architectures, and, in particular, of applications that have a GUI aspect, where both users and the application perform actions concurrently and at arbitrary times and where various components in the architecture must be notified of those actions. Message-based communication is extensively used in distributed environments for which this architectural style is suited.

Central to the architectural style is a principle of limited visibility or *substrate independence*: a component within the hierarchy can only be aware of components "above" it, i.e., components typically closer to the "application," and thus further from the windowing system. Components are completely unaware of the components—including toolkits—which reside "beneath" them.

Substrate independence has a clear potential for fostering substitutability and reusability of components across architectures. One issue that must be addressed, however, is the apparent dependence of a given component on its "superstrate," i.e., the components above it. If each component is built so that its top domain closely corresponds to the bottom domains of those components with which it is specifically intended to interact in the given architecture, its reusability value is greatly diminished and it can only be substituted by components with similarly constrained top domains. For that reason, the C2 style employs the notion of event translation. Each component maintains a mapping from the messages it emits on its top side to those the components above it are capable of receiving on their bottom sides. The C2 design environment [RWMT95] is intended to provide support for building and maintaining these messages.

Each component may have its own thread(s) of control, a property also suggested by the asynchronous nature of

FIGURE 1. A sample C2 architecture (C are components, B are connectors). Jagged lines represent the parts of the architecture not shown.



tasks in the GUI domain. It simplifies modeling and programming of multi-component, multi-user, and concurrent applications and enables exploitation of distributed platforms. Note that separating components into different threads of control is not a requirement. Moreover, a proposed conceptual architecture is distinct from an implementation architecture, so that it is indeed possible for components to share threads of control.

Finally, there is no assumption of a shared address space among components. Any premise of a shared address space would be unreasonable in an architectural style that allows composition of heterogeneous, highly distributed components, developed in different languages, with their own threads of control, internal structures, and domains of discourse

5.0 The Potential for a Chiron-2 Components Marketplace

Having just described the C2 architectural style, we now examine its potential for becoming the basis of a user interface components marketplace by tracing how the features of C2 meet the requirements given in Section 2.

1. Multiple component granularities

   The C2 architectural style implicitly supports multiple component granularities. Nothing within the style prevents arbitrarily large components from being constructed, nor does it prevent arbitrarily small components, so long as they can communicate with connectors. The C2 style also supports the use of a C2 architecture within a single component, and hence composable components are also supported.

2. Substitutability of components

   Due to the substrate independence of components within the C2 style, C2 components have the potential to be highly substitutable. The C2 style also supports the notion of a domain translator which can assist the substitution of components when a particular component does not exactly provide the desired range of capabilities. The C2 style supports substitution of components independent of the number of components below it in an architecture.

3. Parameterizable components

   The C2 style expects that components will be parameterizable at least to the extent that multiple mappings between a conceptual architecture and a particular implementation architecture are possible. For example, a component might be parameterized so it can have its own thread of control, or combined with other components into a single thread of control. Another potential parameter is the processor type of the component, so that one conceptual component can map to one of several binaries as the implementation component.

4. Customizable components

   There is nothing inherent in the C2 style that would prevent the customization of components. However, there is at present no explicit support for customization, since it is outside the set of concerns of the C2 style. A designer wishing to add customization capabilities to a C2 component could have it present a customization interface upon request. This customization interface would be expressed in an abstract, toolkit independent fashion by the component, converted into the appropriate drawing commands by the graphics toolkit, and then presented to the user.

5. Components should be independent of programming language

   The C2 style provides strong support for components written in multiple programming languages. The C2 assumption that there does not need to be a shared address space among components, combined with the ability to have components in multiple simultaneous processes provides sufficient freedom to support components in multiple programming languages.

6. Component-specific help

   Like the customization of components, there is nothing in the C2 style which explicitly supports this requirement. However, a designer wishing to add built-in help facilities to their components could establish a convention that requires them to export, upon request, either help text, a help interface, or a notification to perform a link traversal to a hypertext help system.

7. Support for composing component-specific user interface widgets

   The C2 style supports composable component-specific user interfaces. An artist component may emit messages abstractly informing components lower in a C2 hierarchy how it expects its interface should look. Components lower in the hierarchy may then translate these messages, making the interface description more concrete. Finally, a toolkit may receive these messages and translate them into the drawing commands to place a component's interface onto the user's display along with the interfaces of other components.

8. Easy distribution of components from seller to buyer

   Components in the C2 style will be capable of being distributed as binaries, and can hence be distributed on either a CD-ROM, a tape or disk, or via a network connection. The C2 architectural style thus supports the distribution of components in all of the current major distribution media.

9. Support for multiple sales models

   There is nothing implicit in the C2 style which binds it to a particular sales model, however there is at present no explicit support for any sales model. The C2 archi-

tectural style has the capacity for supporting a class of license handling components which could be used by components upon start-up to verify that one of several floating licenses for that component were available. This would provide support for the workstation sales model, where one component could be used by potentially many people, with a maximum number of simultaneous users. Alternatively, it would also be possible to create a "usage informer" component which would inform a remote server each time a component was invoked, thus supporting a pay-per-usage sales model.

From this examination, it can be seen that the Chiron-2 architectural style either meets, or could potentially meet all of the requirements to be a basis for a user interface components marketplace. However, given the number of requirements which are not completely met, it may be premature to conclude with certainty that the C2 style can become a successful marketplace. The results from this analysis are promising, however.

6.0 Conclusions

We have presented requirements which an architecture should meet for it to be a suitable basis for a software components marketplace. Our examination of the existing software component marketplaces of Unix filters and Visual Basic components suggests that the commercial success of Visual Basic can be attributed to its focus on providing two graphical interfaces for each component: one for use by the end-user, another for use by the designer. They also suggest the non-commercial nature of the Unix filters marketplace can be attributed to the combination of the power and flexibility of the architecture combined with the high skill levels required to acquire and use new components. Analysis of the Chiron-2 architecture indicates that it has promise as a basis for user interface components marketplace. These findings, motivated from an examination of how well they meet the requirements in Section 2, suggests that these requirements have utility for analyzing current architectures, and providing design guidance for new architecture designs.

7.0 Acknowledgments

The authors would like to thank Peyman Oreizy for his helpful discussion and insight, and the Chiron-2 research group for their feedback and encouragement.

References

[AAG93] Abowd, Gregory and Allen, Robert and Garlan, David, "Using Style to Understand Descriptions of Software Architecture," in *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Los Angeles, 1993, pages 9-20.

[ADAG92] Domain-Specific Software Architecture Engineering Process Guidelines. ADAGE-IBM-92-02, Version 2.0 (An abbreviated version of this paper appeared as "A Domain-Specific Software Architecture Engineering Process Outline" in *Software Engineering Notes (SEN)*, Vol. 18, No. 2, April 1993).

[Aitk93] Aitken, Peter G., *Microsoft Guide to Visual Basic for MS-DOS: the Complete Guide to Visual Basic Programming*, Microsoft Press, Redmond, Washington, 1993.

[BaOM92] Batory, Don and O'Malley, Sean, "The Design and Implementation of Hierarchical Software Systems with Reusable Components," *ACM Transactions on Software Engineering and Methodology*, Vol. 1, No. 4, October 1992, pages 355-398.

[BoSc92] Boehm, B. and Scherlis, W. L., "Megaprogramming," in *Proceedings DARPA Software Technical Conference*, Meridian Corp., Arlington, VA, 1992, pages 63-82.

[Haye94] Hayes-Roth, Frederick, "Architecture-Based Acquisition and Development of Software: Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA) Program," Teknowledge Federal Systems. Version 1.01 February 4, 1994.

[McIl69] McIlroy, M. D., "Mass-Produced Software Components," in *Proceedings of the 1968 NATO Conference on Software Engineering*, Garmisch, Germany, 1969, pages 138-155.

[MoFe93] Morris, Charles R. and Ferguson, Charles H., "How Architecture Wins Technology Wars," *Harvard Business Review*, March-April 1993, pages 86-96.

[RWMT95] Robbins, Jason E., and Whitehead, E. James Jr., and Medvidovic, Nenad, and Taylor, Richard N. "A Software Architecture Design Environment for Chiron-2 Style Architectures," Tech. Report Arcadia-UCI-95-01, U.C. Irvine, Irvine, CA, January, 1995.

[Trac91] Tracz, W., "A Conceptual Model for Megaprogramming," Software Engineering Notes (SEN), Vol. 16, No. 3, July 1991, pages 36-45.

[TMA+95] Taylor, Richard N. and Medvidovic, Nenad, and Anderson, Kenneth, and Whitehead, E. James Jr., and Robbins, Jason E., "A Component and Message-Based Architectural Style for GUI Software," to appear, Proceedings of the International Conference on Software Engineering 17, Seattle, WA, April 1995.

[Udel94] Udell, Jon, "Componentware," *Byte Magazine*, May 1994, pages 46-56.