A METAMODEL FOR THE RUNTIME ARCHITECTURE OF AN INTERACTIVE SYSTEM

The UIMS Tool Developers Workshop*

1.0 INTRODUCTION

Developers of interactive systems must make difficult engineering trade-offs in order to optimize their development processes and end products. The trade-offs are made among desirable, but sometimes conflicting, goals such as minimizing the future effects of changing technology and improving system runtime performance. The purpose of this paper is to provide developers with a framework for understanding these tradeoffs and for helping them define and evaluate candidate runtime architectures.

---

* This paper is the product of a number of meetings of user interface tool developers. We found available models of the runtime architecture of an interactive system inadequate for our purposes, and we defined both a model and a metamodel. This work was discussed at the *CHI '91 UIMS Tool Developers' Workshop* (see the companion paper in this issue) and presented at the CHI '91 Special Interest Group, *User Interface Developers' Workshop Report: Seeheim Revisited.* The discussion below is an updated version of the work presented in those sessions. The contributing authors are: Len Bass, Software Engineering Institute; Ross Faneuf, Digital Equipment Corporation; Reed Little, Software Engineering Institute; Niels Mayer, Hewlett-Packard Laboratories; Bob Pellegrino, Digital Equipment Corporation; Scott Reed, V. I. Corporation; Robert Seacord, Software Engineering Institute; Sylvia Sheppard, NASA; and Martha R. Szczur, NASA. For more information, contact the authors at: chiuidt@sei.cmu.edu.

1.1 Models and Metamodels

A number of models exist for the design of user interface systems. Some of the more familiar ones are the Seeheim model (Green; Pfaff & ten Hagan), the Seattle model (Lantz et al.), the Lisbon model (Duce et al., eds.), and MVC (Krasner & Pope) and its more modern successor PAC (Coutaz). A common approach for developing such models is to examine the functionality of an interactive system, decide that separating the user interface functionality from other functionality is the most important design goal, and derive an architecture that supports this separation. The product of this approach is a prescriptive model for an interactive system. Although prescriptive models are desirable, it seems clear from the variety of prescriptive models available and the lack of consensus about which is the best that a single, prescriptive model to fit all types of interactive systems is very difficult, if not impossible, to define.

Coutaz and Balbo (1991) take a different approach. They do not propose a particular architecture. Instead, these authors examine the nature of the data that passes between the user interface and the non-user-interface portions of an interactive system.

During a series of workshops in 1990 and 1991, we expanded upon the approach of Coutaz and Balbo. We began by analyzing the data exchanges and the internal functions of interactive systems. This analysis led to the definition of a model of the runtime architecture of an interactive system, the Arch model. The model was tailored to satisfy our particular goal of minimizing the future effects

of changing technology (e.g., buffering the remainder of the system from the effects of evolving Interaction Toolkits).

Although this particular architectural model satisfies our goal of buffering from the effects of change, it does not satisfy other goals (e.g., maximizing runtime performance). In fact, we contend that no single architecture will satisfy all of the possible goals that developers can have in designing interactive systems. Thus, we introduce the idea of a metamodel with two purposes: to derive any number of architectures, depending upon a particular developer's goals, and to evaluate any proposed architecture in terms of desired goals.

In the remainder of this paper we define some terminology, list potential design criteria, consider the functionalities essential to an interactive system, and present the Arch model and the Slinky metamodel.

1.2 Terminology

The term "application" has several meanings. We define an *application* to be the total system that is developed for its end users. An *interactive application* consists of application domain software and user-interface software. The *application domain,* or simply the *domain,* is the field of interest of, and reason for, the application (e.g., producing a payroll or calculating the orbit of a satellite).

A *user interface runtime system* (UIRS) is defined to be the run-time environment of the interactive application. A *user interface toolkit* is a collection of interaction object classes employing specific interaction media with associated management capabilities (e.g., Motif™ and OpenLook™).

2.0 CRITERIA FOR DESIGN

Design goals can be expressed in terms of criteria for evaluating either the end product or the development process. The following list is not exhaustive but does include some of the more common criteria to be considered when designing an interactive system:

- target system performance:
  - size
  - speed
- productivity of system development tool(s)
- buffering from changes in:
  - application domain
  - medium
  - hardware platform
  - interaction toolkit
- conceptual simplicity
- reuse of code
- meeting functional requirements
- quality of resulting user interface
- cost of:
  - development system
  - target system
- complexity of specification
- time for target system to reach its end user (or the commercial marketplace)
- developers' abilities
- target system extensibility
- adherence to standards
- compatibility with other systems
- complexity of application data
- complexity of dialogue requirements

A given architectural design will satisfy one set of criteria; other designs will emphasize the satisfaction of other criteria. A developer can rank the criteria, determining which are most important to his goals. (Some of the criteria are not affected by architectural considerations; they are included for completeness.)

3.0 FUNCTIONALITY OF AN INTERACTIVE SYSTEM

Regardless of the criteria selected as most important, all designs must provide a certain set of functionalities. This section discusses the data representations and the functions necessary in an interactive system.

There are at least three types of data representations used in an interactive system: representations of the domain data (e.g., names and types in a data base system), representations of the media used for input/output (e.g., pixels for the output to a bit-mapped system) and intermediate representations (e.g., "tabular, labelled, two column data with single entry selection"). Data flows bi-directionally between the domain-specific parts of the system and the input/output media. Data that are expressed in the domain representation are changed (via reorganizations, transformations, aggregations, decompositions, additions, and/or deletions) until at some point they have media representations, and data that are expressed in the media representation are changed to have internal representations.

Several types of actions require control and sequencing. There is the sequencing of end-user, task-level actions, the sequencing of domain-specific actions, and the sequencing required for the input/output media.

With these high-level requirements in mind, we consider the following to be necessary operations for every interactive system:

- control, manipulate and retrieve domain data and perform other domain tasks
- reorganize domain data for user interface purposes
- provide task level sequencing
- provide multiple view consistency
- make media decisions
- choose the interaction objects
- provide physical interaction with the end user
- convert between domain formalisms and user interface formalisms

Different systems place different degrees of emphasis on each function, but every interactive system must perform these functions in some manner. There are, of course, other functions that are important in a modern, well designed system (e.g., detection and reporting of semantic errors, providing help messages), but some interactive systems exist without providing these additional functions.

4.0 THE ARCH MODEL AND THE SLINKY METAMODEL

Of the criteria listed in Section 2, buffering an operational system from changes in technology was selected as most important in our discussions. After listing the functions described in Section 3, we developed a model architecture to insure that our criterion would be met. The next section discusses the model.

4.1 The Arch Model

User interface developers sometimes find both the application domain functionality and the UI toolkit(s) to be existing constraints upon the development of a user interface. The user interface software must manage the interaction between these two externally-provided components (e.g., between a DBMS and X Windows). For this reason, the domain software and the UI toolkits form the two bases of the Arch model. Other essential functionality is provided by three additional components, the Dialogue, Presentation and Domain-Adaptor components. We defined these five components of the model by allocating the functionalities enumerated in Section 3:

Domain-Specific Component - controls, manipulates and retrieves domain data and performs other domain-related functions.

Interaction Toolkit Component - implements the physical interaction with the end-user (via hardware and software).

Dialogue Component - has responsibility for task-level sequencing, both for the user and for the portion of the application domain sequencing that depends upon the user; for providing multiple view consistency; and for mapping back and forth between domain-specific formalisms and user-interface-specific formalisms.

Presentation Component - a mediation, or buffer, component between the Dialogue and the Interaction Toolkit Components that provides a set of toolkit-independent objects for use by the Dialogue Component (e.g., a "selector" object that can be implemented in the toolkit using either a menu or radio buttons). Decisions about the representation of media objects are made in the Presentation Component.

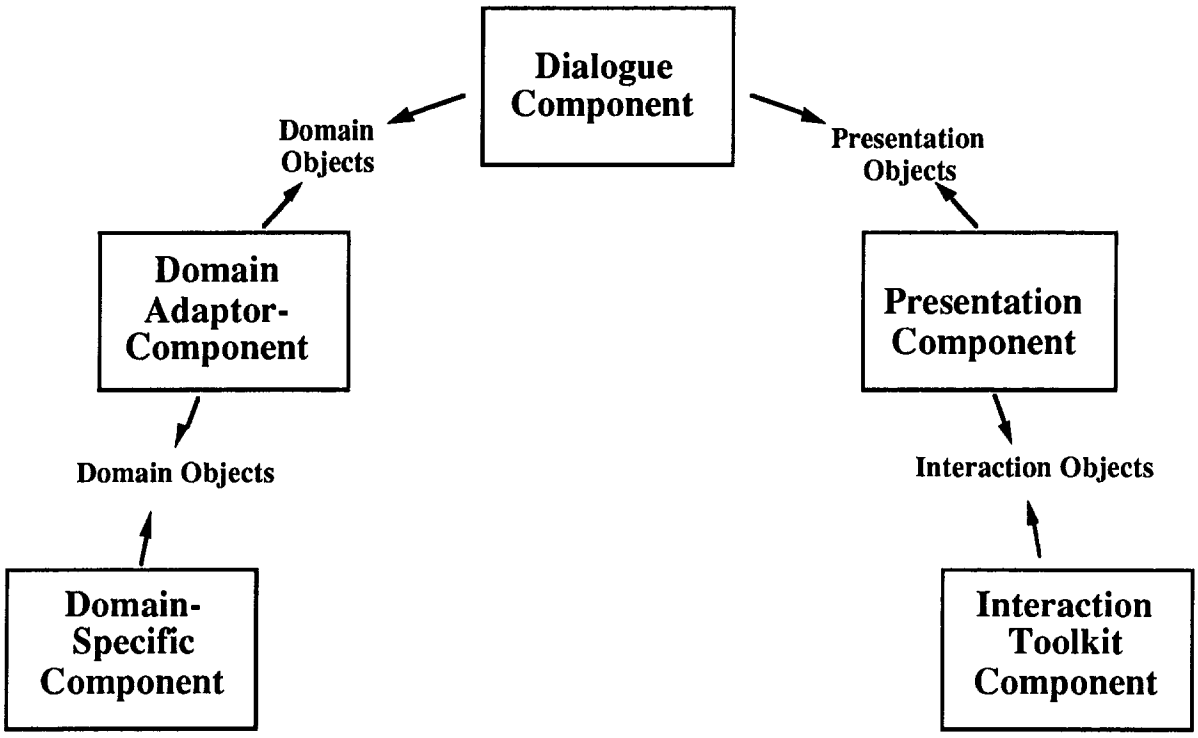

Figure 1. The interfaces between the components

Domain-Adaptor Component - a mediation component between the Dialogue and the Domain-Specific Components. Domain-related tasks required for human operation of the system, but not available in the Domain-Specific Component, are implemented here. The Domain-Adaptor Component triggers domain-initiated dialogue tasks, reorganizes domain data (e.g., collects data items in a list), and detects and reports semantic errors.

Figure 1 shows the components and the types of objects that cross the boundaries between them. Note that the term "object" is used to indicate the formalism that transmits information between the components. The term does not refer to the formal, instantiated objects defined for object-oriented developments. Rather, objects are used here as an expository abstraction for describing communication mechanisms.

Domain Objects are employed by both the Domain-Specific and the Domain-Adaptor Components, but instances of these objects are created by the two components for different purposes. In the Domain-Specific Component, Domain Objects employ domain data and operations to provide functionality not associated directly with the user interface. In the Domain-Adaptor Component, domain data and operations are used to implement operations on domain data that are associated with the user interface. For example, one domain-specific operation of a DBMS would retrieve a set of employee names and salaries by gender from a data base. Iterative review of the list to display parts of succeeding records might need to be done in a Domain-Adaptor Component. Here the Domain-Adaptor Component would supplement the functionality of the Domain-Specific Component by providing a service related to the presentation of information.

Presentation Objects are virtual interaction objects that control user interactions. Presentation Objects include data to be presented to the user and events to be generated by the user. The medium used in the presentation or event generation is not defined. An example of a Presentation Object for use with the list of employees and salaries is "tabular, labelled, two column data with single entry selection".

Interaction Objects are specially designed instances of media-specific methods for interacting with the user. Interaction Objects are supplied by the Interaction Toolkit software and may be primitive (e.g., graphics and keyboard device drivers) or complex. An Interaction Object corresponding to the Presentation Object cited in the paragraph above is a dual bank of radio buttons (which allows the user to select an employee with a particular salary from the "male" column or the "female" column).

4.2 Shifting Functionalities and the Slinky Metamodel

The coupling of functionalities in the components of the Arch model described above was designed to minimize the effects of future changes in the interaction toolkit, the user interface dialogue or the application domain. Dissimilar functions were assigned to separate components in order to allow the modification of one type of functionality with minimal impact on other components in the system. One result of this allocation is that the model does not satisfy all of the criteria listed in Section 2. As discussed above, a model derived to minimize the effects of changing technology may have an adverse effect on the speed of the runtime system. A single model cannot satisfy conflicting criteria.

We need an architecture that can be tailored to emphasize the criteria of choice. The Arch model can be generalized for this purpose. The term "Arch" suggests the more stable development environment that occurs when goals have been set and choices have been made. When we generalize the Arch model, we refer to it as the Slinky metamodel. The metamodel provides a set of Arch models, as opposed to one particular model (Figure 2).

The term "Slinky" was selected to emphasize that functionalities can shift from component to component in an architecture depending upon the goals of the developers, their weighting of development criteria, and the type of system to be implemented. This concept is loosely represented by the flexible Slinky™ toy. (When in motion, the Slinky toy has a dynamically shifting mass.) The graphical representation of the Arch models with varying size components in Figure 2 is meant to convey the concept of models with components that contain varying amounts of functionality.

To clarify the concept of shifting functionalities, consider an example where a function in a Domain-Specific Component was later implemented in an Interaction Toolkit. The Unix file system was originally considered a specific application domain, with file operations such as "open" and "delete". When the interaction toolkit became more sophisticated, a file selection widget was included in the toolkit, thus shifting the functionality from one end of the model architecture to the other.

Consider a second example where the requirement is to take the temperature of a vessel and display the value to the end user graphically and textually. We could allocate the functions as follows:

Domain-Specific Component: Sense temperature of vessel in Kelvin.

Domain-Adaptor Component: Create temperature object.

Dialogue Component: Request temperature from Domain-Adaptor Component, and create presentation object with value units.

Presentation Component: Present value as slider and present units as text.

Interaction Toolkit: Display toolkit objects.

There are other ways to allocate these functions, representing a shift in the functionality assigned to the components. The reallocation is left as an exercise for the reader.

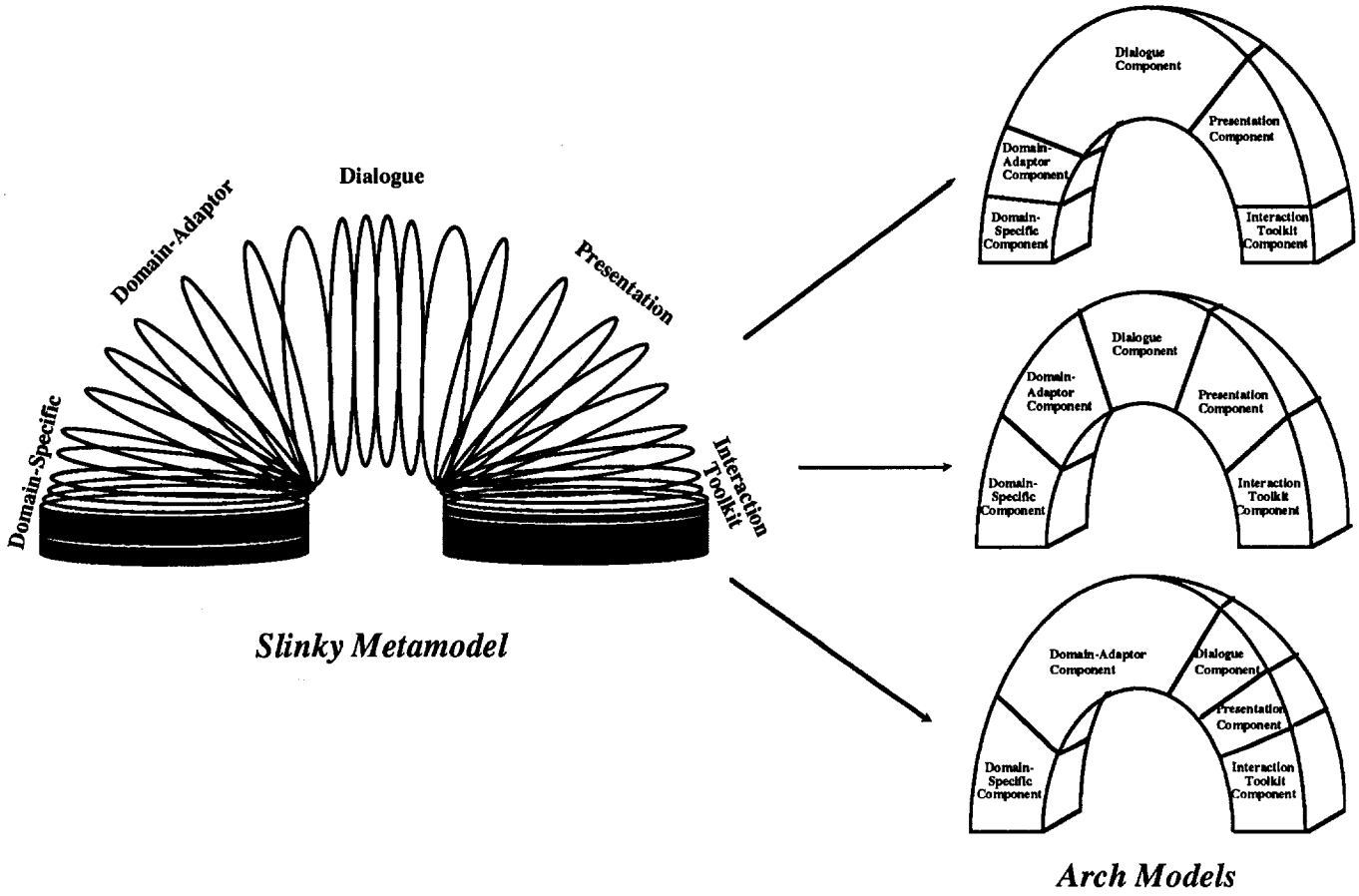

Figure 2. Derivation of the Arch Models from the Slinky Metamodel

4.3 Shifting Emphases

In addition to shifting functionalities, Arch models can vary because of the type of application. Applications with complex domain data will have more capabilities to store, retrieve, reorganize, and transport data among the components of the architecture. Applications with complex user interaction requirements will have more capabilities that support task sequencing, management of multiple views of data, and responses to user actions. All user interface runtime systems must provide capabilities for both data manipulation and user interaction, but most systems emphasize one type more than the other.

Data-oriented systems have extensive facilities for managing information flow, often with minimal dialogue capabilities. These systems typically concentrate on the information being viewed and manipulated by the user. They often have standard mappings from data records to interface representations. These systems are characterized by:

- complex data structures as a major portion of the objects that cross the boundaries between components,
- extensive facilities for specifying structural transformations and for naming and retaining the identity of data aggregates and their elements, and
- facilities (such as a data definition language) for extending the set of data types and for mapping easily to popular database systems or data dictionaries.

Much of the functionality in these systems is concentrated in the Domain-Specific and Domain-Adaptor Components. Examples of data-oriented tools include commercial, forms-based products (particularly those with workstation interface support) and 4GL systems for accessing and maintaining commercial databases (e.g., OMNIS 5).

Dialogue-oriented systems have extensive capabilities for mapping user actions into the behavior of the interface - managing windows, controlling appearance, choosing different techniques for representing the same information, etc. Dialogue-oriented systems are characterized by:

- the provision of a specialized language, production system, or equivalent for mapping user events into domain actions, where the language syntax and expressive focus are oriented towards human/computer interaction,

- a tendency to model domain events in the same terms as user events, and
- simple data interchange models (e.g., the use of shared program variables for communication).

These systems focus primarily on the Dialogue, Presentation, and Interaction Toolkit components of the UIRS. Examples of tools to support dialogue-oriented applications include most of the current Unix-based UIRSs, such as Serpent (Bass & Coutaz) and TAE (Szczur).

4.4 Applying the Slinky Metamodel

The Slinky metamodel provides a context in which interactive architectures can be discussed in terms of desired criteria. Two ways of using the metamodel are : a) to derive an Arch model based on a desirable set of criteria and b) to evaluate a candidate architecture to determine if it satisfies a particular set of desirable criteria.

To construct an Arch model, select the criteria of interest, identify the needed functionalities, identify the interaction objects, and decide how to package the functions into components. Each design criterion enumerated in Section 2 can be analyzed for its architectural implications in terms of the functionalities enumerated in Section 3. Although this is a subjective process, it's not a difficult one. There are known effects for many of the criteria. For example, for the Arch model where the overriding criterion is the efficient and robust management of change, modular separation of functionality is important because of the need for the developers to deal with the continual, rapid infusion of new technology over the life cycle of an application. If system performance is the overriding criterion, one can imagine less emphasis being placed on Dialogue and Domain-Specific Components. Rather, data values may be passed more directly from the Domain-Specific Component to the Presentation Component.

The Slinky metamodel may also be used as an evaluation mechanism for a proposed architecture. The process is to select a criterion, derive the appropriate Arch model, and then compare the distribution of functionalities within the Arch model to the distribution of functionalities in the proposed architecture. If the functionalities of the proposed architecture have components that cross the boundaries of the Arch components, the proposed architecture is not suitable for the satisfying the criterion.

Although we have made some progress in this activity, it clearly is not finished. Future plans include the development of a matrix allocating functionality to the components of the metamodel for each criterion and a discussion of the tools needed to create and maintain each component of the architecture.

5.0 SUMMARY

Developers of interactive systems often have conflicting software engineering goals. In order to make informed judgements about the trade-offs involved in meeting these goals, a software designer needs: an understanding of all of the functionalities involved in an interactive system, an understanding of the significance of his goals in terms of the runtime architecture, and an understanding of how a proposed allocation of the functionalities supports the desired goals. This paper provides guidance for developing and evaluating user interface architectures in terms of developers' goals and the functionalities required.

6.0 REFERENCES

Bass, L. & Coutaz, J. (1991). *Developing Software for the User Interface*. Reading, MA: Addison-Wesley.

Coutaz, J. (1987). PAC, an implementation model for dialog design. *Proceedings of HCI Interact '87*, 431 - 436. Amsterdam: North-Holland.

Coutaz, J. & Balbo, S. (1991). Applications: A dimension space for user interface management systems. *CHI '91 Conference Proceedings*, 27-32. New York: ACM.

Green, M. (1985). The University of Alberta user interface management system. *Computer Graphics*, *19*, 3, 205-213.

Krasner, G. E. & S. T. Pope. (1988). A cookbook for using model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object Oriented Programming*, *1*,3, 26-49.

Lantz, K. A., Tanner, P. P., Binding, C., Huang, K. T., & Dwelly, A. (1987). Reference models, window systems and concurrency. *Computer Graphics*, *21*, 2, 87-97.

OMNIS 5 Application Designers' Handbook for IBM and Compatibles. (1990). Foster City, CA: Blyth Software.

Pfaff, G. & ten Hagan, P. J. W. (1985). *Seeheim Workshop on User Interface Management Systems.* Berlin: Springer-Verlag.

Szczur, M. R. (1991). TAE Plus: a NASA productivity tool used to develop GUIs. *Proceedings of the AIAA Computing in Aerospace Conference* (in press). Baltimore, MD: AIAA.

User Interface Management and Design (1991). In D. A. Duce, M. R. Gomes, F. R. A. Hopgood, J. R. Lee (eds.). *Eurographic Seminars*, 36-49. Berlin: Springer-Verlag.