CHAPTER 4

Software Architecture Models

4.1 Introduction

This chapter demonstrates how one can use analysis of software architectures to generate software designs that are compatible with a chosen 'property profile'. Such a profile must be determined during requirements specification. The approach used in this chapter is to take each external and internal property, and describe (in)compatibilities between it and some interactive software architectures. Architectures developed and refined during the system and software design phases can be compatible with this profile in four ways.

1. The property is *delivered* without further developer effort.

2. The property can be *assessed* with developer effort (perhaps considerable) and skill (always extensive).

3. The property can be *addressed, assisted* or *measured* but cannot be assessed immediately.

4. The property is *not impacted* – the architecture does not interact with the property.

The last form of compatibility is *neutrality*, but an architecture will not be neutral with respect to all properties. Were this so, there would be no reason to use it (or the list of properties presented in this book is not complete!). The first three forms of compatibility are *support* to varying degrees. An architecture is said to be compatible with a property if it provides it automatically, supports it, or is neutral with respect to it.

During early development activity, desired properties should be selected and be allocated a priority weighting. The designer must accept these constraints during the subsequent architectural design phase. The actual process of defining and weighting properties, when coupled with the analyses developed in this chapter, can guide the design of architectures which are compatible with system requirements. These analyses are similar to those given in previous chapters, but rather than examine interactions between software techniques and quality properties in general, we restrict our attention to architectural models.

C. Gram et al. (eds.), *Design Principles for Interactive Software*

4.2 A Framework for User Interface Software Architectures

The need for architectures arises in response to complex functionality. A system's functionality is expressed as a set of capabilities ('what the system can do'). These capabilities can generally be regarded as operations on some model of an application domain, which may, for example, transform (some of) the model, or perform calculations on it.

A simple domain may only require a few operations. These could be readily understood as comprising the system's overall behavior. Typical application domains are more complex, with many operations, and comprehension must be assisted by grouping operations on the basis of similarities. These groups must be readily understood as comprising the system's overall behavior. Functionality is thus coarsened to make it manageable.

Such a grouping of the operations for an application domain is called a *functional partitioning*; this is one of the starting points for the design of a software architecture. Architectural design for software systems involves (at least) two other factors: its structure and the allocation of domain function to that structure (Kazman *et al.*, 1994). Architectural structure will be considered first.

In order to work together as a system, coarse decompositions must be recomposed by linking function groups together. The function groups must then be allocated to some architectural structure. There are two kinds of entity in this structure:

- a collection of components which represent computational entities (e.g. modules, procedures, processes or persistent data repositories);
- a representation of the connections between the computational entities, i.e. the communication and control relationships among them.

The relationships between the components must provide for efficient 'vertical' abstractions over several components, e.g. widgets composed of functions from several components.

Each allocation of function to structure should provide the designer with a different understanding of the realization of function in a software system. However, the key motivation for architectural analysis is not the creation of such understanding. Rather its purpose is to support rational choice between alternative software architectures (by comparing the corresponding allocation of function to structure in each one). Such comparisons can be guided by the probable support that each structure can deliver for desired properties for a proposed system.

4.2.1 Functional Partitioning

A *functional partitioning* is a grouping of the operations for an application domain. Here, the application domain is understood as the general one of

computer systems interacting with humans. Before proposing a functional partition for this domain, it may help to make an obvious statement:

> Valid extensive conclusions about the construction of interactive systems can only be drawn if there are extensive commonalities among the systems.

Fortunately, there is extensive consensus on the adequacy of one form of functional partitioning for all interactive systems. It is based on the necessary transformations of information that flows between users and the underlying computations of interactive systems.*

In order to discuss partitioning clearly, untainted by specific features of different architectures, a standard functionality decomposition model is adopted here. The Arch/Slinky metamodel, discussed in UIMS (1992), is used with slightly modified terminology (see Section 4.5.1 for further details). The five groups of functionality used in the model are illustrated in Figure 4.1.

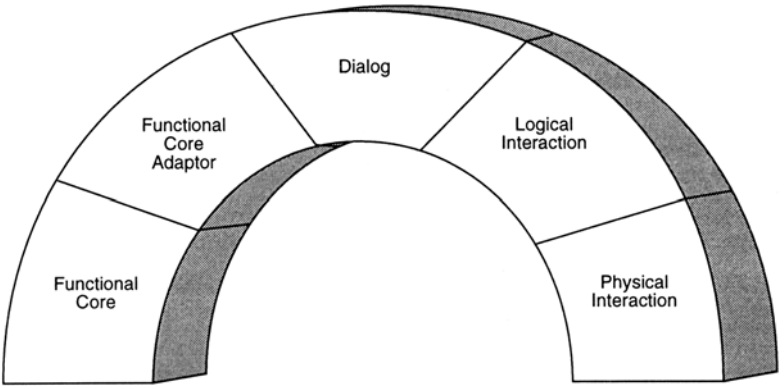

Figure 4.1 *The Arch/Slinky Metamodel.*

- **Functional Core (FC)**: This group of functions implements work domain features. They are also often called the 'application', but that term is ambiguous (Cockton, 1987b).
- **Functional Core Adapter (FCA)**: This group of functions mediates between D and FC by providing more generic work domain concepts. They may aggregate system data into domain-oriented structures, provide a unified interface to heterogeneous FCs, perform semantic checks on data and trigger domain-initiated dialog tasks. Results from the FC are passed through to the D and onward for presentation to the user.

* Other less common partitions use phenomenology (i.e. reflecting a user's decomposition of the system into objects), interaction structures (e.g. steps in a standard interaction cycle) or generalized interactive functions (e.g. help, customization, history) (Cockton, 1987a).

- **Dialog (D)**: This group of functions mediates between domain-specific and presentation-specific functions. It controls task sequencing and context management and ensures consistency (possibly among multiple views of data).
- **Logical Interaction (LI)**: This group of functions mediates between PI and D. It provides a set of logical interaction objects (sometimes called virtual objects), and presentation-specific functions to the dialog.
- **Physical Interaction (PI)**: This group of functions implements the physical interaction between the user and the computer. It deals with input and output devices and is typically realized as a user-interface toolkit and/or a proprietary interface library.

This functional partitioning mirrors nicely the levels of input/output abstractions introduced in Chapter 1. It is based on an information flow *component provision strategy* (Cockton, 1991) reflecting a general observation: that the information flow starts and ends in the physical devices with which users interact and that there is a final 'U-turn' in an interactive system which lies deep in the *underlying functionalities* realizing the semantics of a specific work domain.*

To show the general applicability of this partitioning, a thesaurus is provided, showing how a variety of well-known architectures' specific choices of terms relate to the terms which are used here. This is presented in Table 4.1 for a number of well-known user interface models. For each model, the functions which that architecture instantiates are given equivalents in terms of the partitioning described here. Further information about these systems may be found in Coutaz (1987), Lantz *et al.* (1987), Krasner and Pope (1988), Nigay and Coutaz (1993), Pfaff (1985) and UIMS (1992).

4.2.2 Understanding Functional Partitionings

Using the above classes of functions, an example decomposition for a simple *climate control system* is presented below. The system is to support control of temperature and humidity in a building. A control unit is used to set the desired temperature and humidity. The system monitors temperature and humidity and maintains the desired climate by controling a furnace and air conditioning equipment. The control unit displays two sets of information: the actual temperature and humidity, and the desired temperature and humidity. Functional partitioning within this simple system shall be made to reflect the generic decomposition shown in Figure 4.1.

Functional core functions interface with temperature and humidity sensors, the boiler and air conditioning equipment; store the desired temper-

* 'U-turns' occur when input processing changes to output initiation. There are other 'U-turns' whenever feedback occurs (e.g. cursor tracking as lexical feedback for logical devices).

Table 4.1 *User Interface Functional Partitioning.*

Model	Components	Functional Equivalents
Seeheim	Appl. Interface Model	FCA
	Dialog Control	D
	Presentation	LI + PI
Seattle	Application	FC
	Workstation Manager	D
	Dialog Manager	LI
	Workstation Agent	PI
PAC-Amodeus	Functional Core	FC
	Interface with FC	FCA
	Dialog Controller	D
	Pres. Techniques Comp.	LI
	Low-level Interact. Comp.	PI
Arch	Domain Specific	FC
	Domain Adapter	FCA
	Dialog	D
	Presentation	LI
	Interaction Toolkit	PI
MVC	Model	FC
	View	LI (output only)
	Controller	LI (input only)
PAC	Abstraction	FC
	Control	D
	Presentation	LI + PI

ature and humidity; maintain the desired climate; and report the actual temperature and humidity. These constitute the underlying functionality of the system and can be independent of any user interface functions. They may be supplied by the manufacturers of the control and sensor hardware.

Functional core adapter functions convert information between the formats required by the functional core and those used by the user interface. The functional core handles the desired and actual climate as four separate numbers.

All temperature values are in Fahrenheit, as the functions are written to interface with hardware components that accept and produce digital Fahrenheit values. The user interface displays and receives temperatures in

Celsius, as the controller is designed for markets where Celsius is preferred. The functional core adapter thus includes two functions for converting between Fahrenheit and Celsius. The Celsius to Fahrenheit conversion function is called by a function that implements one of the system's two abstract commands (change desired temperature, change desired humidity).

The functional core adapter also includes a function that forms status 'records' and communicates them to the user interface. Each status message contains a pair of values, a desired setting and a current value. The user interface displays two statuses, one for temperature and one for humidity. As the functional core's reporting functions are not event based, this status forming function polls the functional core periodically to get the current temperature and humidity.

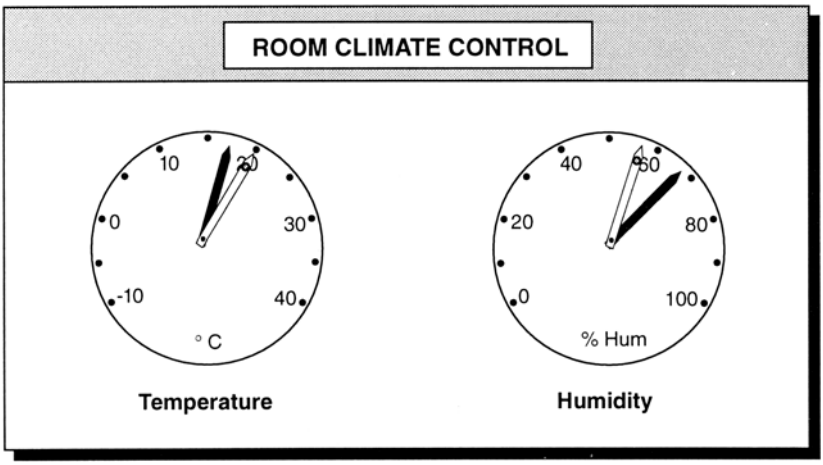

Figure 4.2 *Display for climate control.*

Dialog functions include one that implements the presentation strategy for temperature and humidity status records by converting them to parameters for the two controller-gauge widgets shown in Figure 4.2. Another function responds to a new user setting for desired temperature or humidity by calling an abstract command in the functional core adapter.

Logical interaction functions implement controller-gauge widgets. In the particular PI realization of the interface shown in Figure 4.2, each gauge has two pointers. The black pointer shows the current value and the white pointer the desired value. The white pointer has a circle near the end (the designer's intention is that this should afford manipulability). Logical interaction functions will implement these specialized widgets for a specific set of physical interaction functions. Few existing physical interaction libraries support these controller-gauge widgets. Different sets of logical interaction functions will be provided for each toolkit, window system or graphics

library that provides the physical interaction functions. However, these sets of functions would appear to be identical to any dialog client.

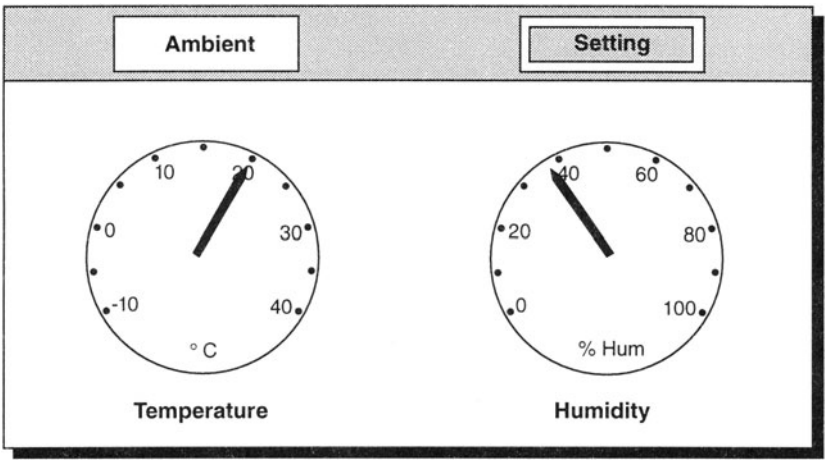

Figure 4.3 *Display with obscure button labels.*

Physical interaction functions interact directly with external peripheral devices. They display logical interaction objects as graphical primitives with requested graphical attributes. They detect user interaction with these objects. One can imagine using a device such as a mouse or a touch screen to alter the temperature.

The generic functional partitioning introduced here thus maps rather nicely onto this simple example. The partitioning can be further illustrated by the scope of changes to the user interface to the climate control system.

Changes to the way in which the data is presented to the user will affect different functional partitions. For example, Figure 4.3 shows a moded dialog with fairly obscure button labels ('ambient' means current, 'setting' means desired). For this design, some functional core adapter function(s) would have to prepare 'climate records' that pair temperature and humidity rather than actual and desired values (as in status records). Dialog functions would have to maintain two modes: in one mode, ambient temperature and humidity are to be displayed as dials; in the other state the current settings are displayed (and can be re-set). The buttons at the top of the display correspond to the two modes. A selected button is highlighted in some way to indicate the current mode. Dialog functions would respond to button presses by changing the mode. Other functions would implement behavior specific to each mode (i.e. changing the displayed information, enabling/disabling user interaction). Logical interaction functions would implement the separate control and view gauges, as well as the mode but-

tons. These functions would provide a portable interface to specific physical interaction functions.

Interestingly, this second design uses an interface which provides an explicitly modal appearance to the user, whereas the previous one was modeless. This change in the manner of operating the controls necessitates a change in the functional core adapter. It should be noted, however, that changes to the adapter could also be needed in the modeless case if the functional core provided climate records rather than actual and desired values (status records).

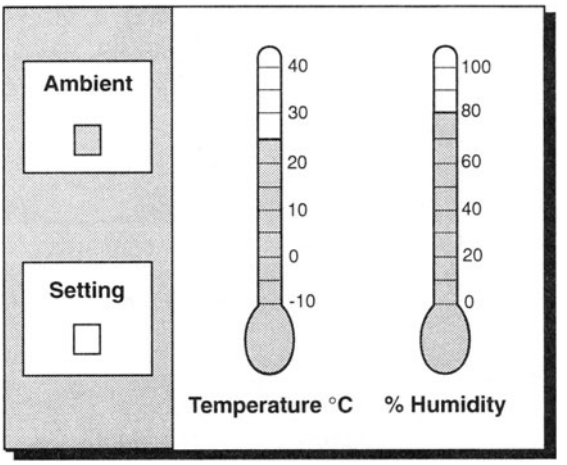

Figure 4.4 *Display with thermometer sliders instead of gauges.*

As a final example, consider the changes required to change the design in Figure 4.3 to that in Figure 4.4. Here only the logical interaction functions need to be changed (in order to render a thermometer slider rather than a dial style gauge). However, the new functions would appear to be identical to the dialog client. To do this, logical interaction functions support concepts such as 'scale' and 'selector', which they then translate into physical interaction objects, such as 'button', 'dial' or 'slider' (Figure 4.2 requires a concept like 'double-scale').

4.3 Architecture and External Properties

It is important to determine which functional partitions need to be considered when attempting to satisfy an external property. This will enable the designer to construct a system with more predictable properties. It also improves the understanding of the interrelationships between internal and external properties, as they are affected by software architecture.

The functional partitioning given in Section 4.2.1 will be used here as

the base reference. The properties are analysed for their relationships to the five canonical partitions of an interactive system – FC, FCA, D, LI and PI – illustrated in Figure 4.1.

This and the following section address the properties that support goal and task completeness, interaction flexibility and interaction robustness. If a property is considered desirable in the context of a given design then it must be considered in this analysis. For each such property it is necessary to determine which of the functional partitions will be impacted when attempting to satisfy the property. The notion is here that the list of especially desirable properties for a system will constrain the designer to certain architectural solutions.

This leads to an important observation: there are two levels of analysis at which the various properties might impact the choice and implementation of an interaction software architecture. A property might pose requirements to the allocation of function to structure, that is, it requires a specific relationship between functional partitions. It can also pose specific requirements to the implementation of run time support for one or more functional partitions. This distinction will be exemplified below. A property is said to impact a functional partition:

- if this partition must necessarily be the focus of attention in an analysis of the system with respect to the satisfaction of the property;
- if the satisfaction of this property requires extra functions to be implemented within this partition than would be the case if the property was deemed irrelevant to the design.

The discussion of how properties impact architectures is followed – in Sections 4.5 and 4.6 – by a discussion of examples of both conceptual architectural models and more implementation-oriented architectures in this context. Finally, in Section 4.7 a specific Chiron-1-based system (the climate control system described above) is assessed with respect to support for a list of given properties.

4.3.1 Goal and Task Completeness

The architectural interactions with the principle of completeness are mediated by the internal property of *functional completeness* defined in Chapter 2. The functionality required to support adopted goals and user tasks must be established early in design, and thereafter the issue becomes one of faithful realization rather than correct determination. There are thus no direct interactions between this principle and architectural models.

4.3.2 Interaction Flexibility

The analysis below examines the impact of external properties related to interaction flexibility on functional partitionings.

Role Multiplicity

This property is most visible in the dialog component. Different roles imply different dialogs handled in various subdialog components that communicate with the same functional core, or different functional cores. The overall management of these – metadialog control – has to be handled in the dialog component. In order to properly support this property, the dialog should allow decomposition into sub-dialogs. For example, in the PAC-Amodeus conceptual software architecture (described in detail in Section 4.5.3) the dialog controller is organized as a hierarchy of PAC agents (Nigay and Coutaz, 1993). To provide for the Role Multiplicity property, a metadialog mechanism which communicates with the control part of each PAC agent is needed. This also is similar to the fusion mechanism used for implementing multi-modal systems in PAC-Amodeus by Nigay and Coutaz (1995). See also Figure 4.5.

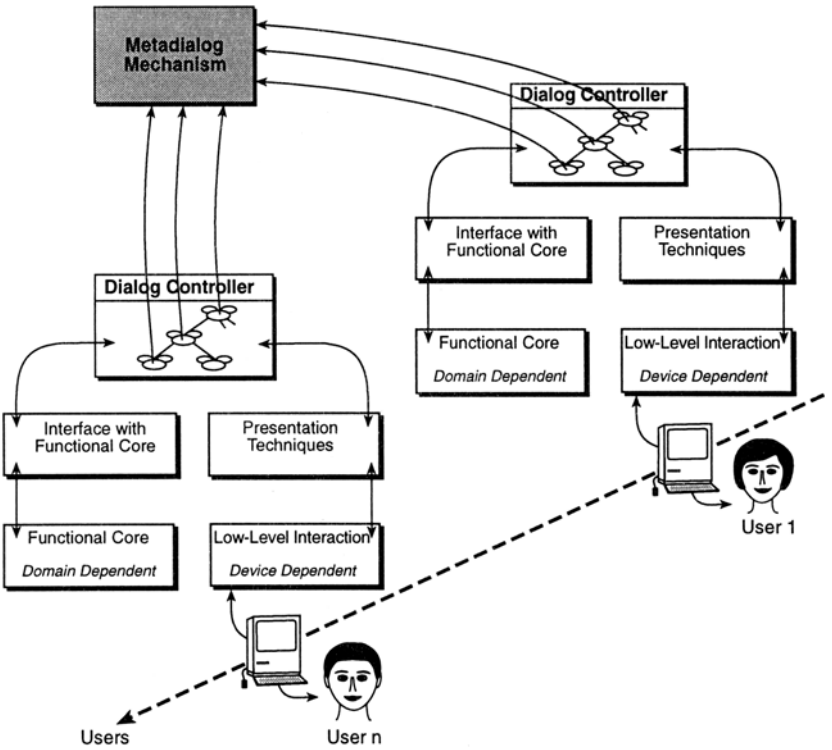

Figure 4.5 *Human role multiplicity: Applying PAC-Amodeus.*

Non-Preemptiveness

This property is most visible at the dialog level. In addition, the feedback needed to make pre-emptiveness perceivable is at the dialog level too.

For example, when trying to print a document on a Macintosh, if there is not enough memory left to print the document, a dialog box appears to indicate this fact, but the system does not allow the user to do anything else to enable printing until the dialog box is cleared. The system is pre-emptive. The Dialog defines the task sequencing, so it is clear that this component is the one that must be addressed to solve this problem.

In verifying that an interactive system is non-preemptive, the designer is greatly aided if the dialog aspects of the system are architecturally isolated. A separate dialog component will isolate the concerns about pre-emptiveness from other concerns of the system; it also isolates the point where verification of this property must be performed.

If an architecture lets dialog management be distributed – e.g. across an object space like PAC (Coutaz, 1987), MVC (Krasner and Pope, 1988) or ALV (Hill, 1992) – then it is probably easier to implement non-preemptiveness. However, it is more difficult to analyse distributed models for non-preemptiveness than if the dialog management is centralized. At the system component level, there might be different strategies for representing dialog control – some better suited for analysis than others. For example, a dialog controller built around a state transition network model provides support for the creation and analysis of non-preemptive dialogs. A dialog built with a traditional programming language (without state-transition support) would make the creation of non-preemptive dialogs more difficult.

Multithreading

In a system with multithreading the user may engage in several tasks simultaneously. This means that context management is needed when handling multithreading, and this functionality is available in the Dialog component. Hence multithreading and multitasking are strongly related to the Dialog.

If an interactive system has to support the user's desire to pursue multiple threads of interaction addressing separate tasks, then it must be able to preserve the state of any active threads of interaction. This has the following architectural implications.

- The Dialog must record the state of all threads of interaction in order to allow for arbitrary interruption and resumption of conversation threads.
- Although it is not strictly necessary to differentiate the various threads of dialog at the LI level, a reasonable architecture will reflect multiple concurrent dialogs pertaining to different user threads or tasks by means of multiple presentation schemes.

Consider the effect of these on the Dialog component: when attention

is moving from one window to another, the Dialog has to be aware of the context switch (which then results in providing feedback such as changing the cursor or highlighting the selected window).

Reachability

The reachability property concerns a user's intention in going from one state to another. As indicated in Chapter 1, states can be defined at different levels of abstraction. Therefore, from an architectural perspective, reachability affects many functional partitions. Whether the user is forward- or backward-seeking (as defined in Chapter 2), the Functional Core must allow for reachability. This could mean that sufficient history information is preserved in order to undo back to some arbitrary point in the interaction. It also means that the system avoids blind alleys – interactions which leave the FC in a state from which it is impossible to reach other states.

It may be necessary to incorporate functions that keep track of the progress of the user from one state to another in order to be able to backtrack. If there are aspects of the Functional Core state which are not accessible to the user interface then changes inside the Functional Core or Functional Core Adapter are needed to permit backtracking. For example, if the FC does not provide a facility for undoing the results of previous function executions, then the FCA could keep track of changes to the state of the FC and implement undo itself. To permit run time reasoning about forward reachability, the system needs a description of the available functions at any given time. This description could be found in the Functional Core Adapter or the Functional Core.

Since analysis of reachability necessarily has to be done by looking at the dialog description, the functionality of the Dialog component is affected. For instance, a state transition diagram representation of the Dialog may indicate that the user can always return to a particular dialog state (e.g. repeated pressing of escape will return to the main menu of a menu-driven system). It is important to note, however, that this reachability constraint may not adequately reflect the user's intention, as there is no guarantee that returning to the same dialog state has had no effect on the Functional Core. The user will need to know information about the consequences of pressing an undo key in order to determine if that action will satisfy his or her intention.

An issue of forward reachability is how to indicate to the user the availability (or not) of following operations in the current interaction context. This is the property of observability which will be discussed in Section 4.3.3. The Dialog therefore must make reachability information accessible to the user.

It can be seen that reachability involves a close relationship between

functional partitions which seems to violate separation of concerns. Reachability not only has meaning in many of the identified functional partitions (FC, FCA, D and LI) but also implies that capturing the user's intention may require an evaluation within more than one functional partition at a time. For example, the discussion in the preceding paragraph for the dialog component requires an evaluation of the Functional Core and Dialog as well as the information that is communicated between them via the Functional Core Adapter. In conclusion, the present understanding of architectural separation does not localize reachability concerns. Since the property of reachability depends on functions in several partitions, no architectural modeling guarantees easy achievement.

Device Multiplicity

Dialog is independent of device and representation considerations. Therefore, concerns of device multiplicity do not extend beyond the Logical Interaction (LI) and the Physical Interaction (PI). To support device multiplicity, either the LI or the PI implementation must be able to distinguish different interaction device usages and potentially allow for their concurrent use. For instance, PI must provide time stamps so that fusion of I/O streams and time-out strategies can be handled. LI must contain mechanisms for abstracting device-dependent data into device- and representation-independent data to ensure that the Dialog is device- and representation-independent.

Some issues of device multiplicity are similar to multithreading in that two physical devices can be used concurrently.

Representation Multiplicity

This property means that a single application concept may be represented by more than one presentation object, both input and output. For example, the dials in Figures 4.3 and 4.4 represented the same underlying information, but had different appearances and different means of interaction with the user. Similarly, a new temperature or humidity setting might be specified using natural language or a command language.

The LI and PI are necessarily representation-dependent, whereas the Dialog only becomes representation-dependent when moded dialogs are selected for the representation of task methods (this is a temporal representation, but a representation nevertheless, and alternative dialog sequences will produce alternative representations).

Representation multiplicity can be achieved in a number of different ways – it does not necessarily belong to any one function. For example, one could pass a single application object from the FCA to the D, and from the D convert it into two LI objects. Or, the FC could pass a single object to the FCA, which would then split it into two objects for the D. Representation

multiplicity would not occur in the FC, since the application should remain presentation-independent, and it should not occur in the link between the LI and PI components (i.e. having the LI split a single object into two distinct presentation objects), since this should be a one-to-one mapping, for the sake of generality.

As discussed in Section 4.3.2, multiple presentation schemes can be used to effectively support multiple threads of user interaction. In this case, representation multiplicity will be achieved in the D component which is the locus for multithreadedness.

I/O Re-use

Re-use of *input* is the utilization of data previously entered in some current context. Since input re-use is a semantic function, it is ideally supported in the Functional Core Adapter and the Dialog. One of these functional partitions will gather the input (into a history buffer, for example) and enabling re-editing and re-use of it.

Re-use of *output* relies on passing the same structure of information between various partitions of the architecture. In the case of passive output re-use the LI is impacted since it has to accept that the present output representation is returned as input (for example, cut and paste features). Active output re-use is exemplified by live text, see Fraser and Krishnamurthy (1990). An example of a live text application is the ability to edit the output of a spelling checker and have the changes propagated back to the source files. The information required of the spelling checker program is simply the name of the source file(s) and the location of the (possibly) misspelled words. In order to enable active output re-use, two possible solutions present themselves:

- the LI may be modified in order to present a consistent set of input objects to the Dialog.
- the Dialog itself is modified in order to map between input objects and the requested functions.

Of these two solutions, the first appears to ensure separation of concerns – in particular that the Dialog is representation-independent.

Reconfigurability

Different types of reconfigurability may exist within several different functional partitions. Many systems allow the users to define new commands, and this belongs to the dialog level. For instance, the use of Unix shell is a way, at the dialog level, to customize the users' command line interface.

At the LI level, the style of interaction (menus or command-lines or graphical buttons) can be adjusted as may be done using the X Window resource manager. The ability to tailor displays to the user's chosen format

clearly impacts LI; for example, displaying a menu in Kanji as opposed to ASCII may require different logical interaction objects because the two representations are so different. Choosing a different set of key bindings or mouse clicks to invoke a command may impact the LI as well.

At the PI level, the use of initialization files for window managers permits tailoring the look and feel of various interaction objects, such as windows, icons, and pointers. Reconfigurability at the PI level is also available on platforms that support easy addition and substitution of input and output devices (e.g. the Apple Desk Top Bus (Apple, 1991)). Such plug-and-play capabilites interfere with software architecture since critical software support is needed in addition to hardware features.

Lastly, reconfigurability can in some cases be provided by FCA functions, for example to support a 'training wheel' approach as in Carroll and Carrithers (1984) where novice users have restricted access to a system's functionality. Such capabilities would be used by a system administrator rather than by an end-user.

Reconfigurability is defined as the user's ability to adjust I/O forms. This kind of flexibility should not be provided in the functional core because the FC deals with the domain-dependent functions of the interactive system. Changes and flexibility at this level modify the functionality of the system and belongs under the heading of the internal property modifiability in Section 4.4.

Adaptivity

Self-adaptivity is an action on the part of the system to better provide services to the user. Some examples of self-adaptivity are caching information for quicker retrieval, and automatic creation of user profiles. These types of adaptivity exist in the Functional Core, Functional Core Adapter and Dialog.

Migratability

Migratability allows transfer of control between the user and the system for performing some set of tasks. This is closely related to the Dialog component which controls the dialog and manages the sequencing of tasks. Hence the Dialog must be constructed with an ability to switch between various agents performing tasks. Furthermore, the Functional Core (or the Functional Core Adapter) must be able to initiate tasks that are 'migrated' to the system from the user.

As an example, an expert system can allow the user to transfer some decision-making responsibilities to the system and the system can decide when its decision-making process requires further user guidance. The diagnosis and correction of errors in a factory control system may have similar transfer capability between the user/operator and the system.

4.3.3 Interaction Robustness

Robustness is concerned with those features of the interaction that support successful achievement and assessment of the goals. Interaction robustness is divided into the sub-properties discussed in the following paragraphs.

Observability

A system which supports observability allows the user to evaluate its internal state without modifying it. Browsability, on the other hand, allows the user to *explore* the current internal state. This functionality is basically handled within D. However the more detailed exploration of an internal state may require (or use) knowledge from the Functional Core. For example, in the World Wide Web, browsing through the Internet will often cause files to be transferred from remote sites to the user's home site.

Insistence

Insistence deals with the the effect of a communication act. Insistence varies according to the output representation and device (e.g. audio versus graphics). Insistence will ideally be handled by the PI or LI – these will provide control over the properties of each presentation object.

However, in some cases this will affect the dialog component. Consider, for instance, a text editor which provides multiple windows to view the same document. If the user modifies the document, and wants to quit without saving, a 'Quit without saving?' dialog box will appear. This dialog box has to be acknowledged by the user before going on. If there were a single window, this insistence could be handled by the logical interaction component. In the multiple view case, however, insistence would require that *every* window should be locked under these circumstances. The user must then deal with the application query before proceeding. This insistence could be reinforced by beeps when the user attempts other actions. As the presence of multiple windows is known only by the dialog component, it will have to implement that insistence.

Insistence at the PI level is supported by hardware features such as lights on keyboards (e.g. to indicate 'Caps Lock' on). To exploit these features, LI functions must give full access to them, but they are generally abstracted away from the actual rendering device in favour of device independence.

Deviation Tolerance

As noted in Chapter 2, the system should not only help the user recover from errors but also prevent or discourage errors from occurring. However, analysing a system to discover potential error situations usually involves functionality within all the components. An example of this is seen in Dix *et al.* (1993), pages 292–5. A particular word-processor automatically saves

when the program terminates, and the user can set a flag to override this function. However, this default can be overridden by setting a (temporary) flag which allows the user to exit the program without saving the text. Of course, if this flag is accidentally set, the system enters a 'dangerous state' where the user might lose important work. Detecting such 'dangerous states' requires an understanding of the semantics of the system (ideally located in the Functional Core Adapter, with necessary support from the Functional Core). This can be seen in the example, where one needs to know that exiting without saving is a 'dangerous' thing to do. Discovering what user actions can cause these states requires an analysis of the Dialog. Finally, finding whether these sequences are easy to perform accidentally requires an examination at the Physical and Logical Interaction Levels. In the example cited one keyboard design (using function keys) gave no errors at all, but a slightly different design meant that the most common exit sequence could, by a minor typing slip, lead to the dangerous state.

Deviation tolerance for certain input errors can be supported at the LI level. For example, input can be smoothed when digitizing curves. More general provision of deviation tolerance support is found in LI modifications for users with motor difficulties; Apple Macintosh computers provide several facilities (Apple, 1991) for users with special needs (e.g. 'sticky keys', 'slow keys'). This reduces the need for speed and co-ordination of mouse buttons and keyboard modifiers. The capabilities could be regarded as supporting reconfiguration, but the aim is to remove sources of errors.

Predictability

Predictability of an interactive system means that the user knowledge of the interaction history is sufficient to determine the result of the future interaction. It deals with the user's ability to determine the effect of operations on the system. It is a user-dependent concept and is not primarily influenced by the software architecture of the interface.

Honesty

Honesty is the ability of the user interface to provide the user with an observable and informative account of the state changes effected by operations. It is a manifestation of the relation between the internal and external states of the user interface. It is defined in the Dialog which handles the mapping function between the Functional Core and the Logical and Physical Interaction. But honesty can only be obtained if the LI contains the right functionality, i.e. widgets that can produce renderings with sufficient information for the user.

Access control

Access control may affect the Functional Core or the Dialog component or the Logical Interaction component. The component affected is usually the one that manages the pieces of data to which access must be controlled. For instance, write access to a file in a Unix system is handled in the Functional Core, because files are objects of the functional core. Similarly, in a multi-user graphical editor, the locking of graphical objects can be handled in the Dialog component. Finally, in a shared editor, the temporary blocking for user interaction (where the cursor is 'locked') is relevant to the Logical Interaction component.

Pace tolerance

Pace tolerance refers to the temporal properties of the user's interaction with the Functional Core. Examples of pace tolerance are type-ahead (where the user is temporally ahead of the system), and time-outs, where the user is behind the system. Pace tolerance can affect any of the components FC, FCA, D, LI, or PI. For example, type-ahead is implemented by the Physical Interaction component. Video games are instances of applications for which pace tolerance properties are dictated by the Functional Core.

Pace tolerance also exists from the system's point of view. For example many systems provide feedback about partial completion of activities (such as fetching or copying large files).

Temporal properties cannot be demonstrated in any existing software architecture. Multi-agent approaches (as in the PAC-Amodeus model, see Section 4.5.3) can, however, provide partial feedback before the completion of a user command. This is seen as being a good way of achieving pace tolerance.

Tables 4.2 and 4.3 summarize the impact of interaction robustness properties on functional partitionings and show how well each property corresponds to a single functional partitions or adjacent partitions.

Reachability and Pace Tolerance are pervasive. Consideration of these properties cannot be restricted to a few functional partitions, but certain aspects of each property can be localized to each of the functional partitions introduced. Developers should therefore still be able to focus on one partition at a time when considering these properties.

However for the effectively pervasive properties of reconfigurability, I/O re-use and deviation tolerance, only reconfigurability can be factored into features specific to each functional partition. For I/O re-use and deviation tolerance, there will be close coupling between the source of the re-use input or output, or the source of the deviation, and its handling at a more abstract level of interaction. The value of architectural analysis here is its highlighting of two problem areas for software design. Advance knowledge

Table 4.2 *Interaction Flexibility vs Functional Partitioning*

Flexibility Property	Partitions				
Role Multiplicity			D		
Non-Preemptiveness			D		
Multithreading/Multitasking			D	LI	
Reachability	FC	FCA	D	LI	PI
Device Multiplicity				LI	PI
Representation Multiplicity		FCA	D	LI	PI
I/O Re-use		FCA	D	LI	
Reconfigurability		(FCA)	D	LI	PI
Adaptivity	FC	FCA	D		
Migratability	FC	FCA	D		

Table 4.3 *Interaction Robustness vs Functional Partitioning*

Robustness Property	Partitions				
Observability	FC		D		
Insistence			D	LI	PI
Deviation Tolerance	FC	FCA	D	LI	
Predictability					
Honesty			D	LI	
Access Control	FC		D	LI	
Pace Tolerance	FC	FCA	D	LI	PI

of such difficulties can improve development. On the other hand, the architectural analysis using the functional partitions offers no support for the creation of predictable systems.

Several properties interact with three or four of the functional partitions. A certain pattern seems to emerge here: properties such as adaptivity and migratability are mainly tied to semantic features of the system as expressed in the FC and FCA partitions; others – representation multiplicity, reconfigurability, and insistence – are tied to more representation-linked features found in the PI and LI partitions. Developers can focus their attentions on different 'coherent' regions of a software architecture when considering these properties. Interestingly, support for the representation-linked properties seems to be more common in real systems than support for the semantic properties. The former are given adequate to good support from

existing tools and materials (see Chapter 5), but general support for the latter has been developed almost exclusively for research systems.

Access control does not fit into the distinction between representation-linked and more semantic properties as it affects both the FC and the LI partition.

Other representational properties are more focused, such as multi-threading, device multiplicity, and honesty which are supported by two adjacent functional partitions. Observability would also have a tight architectural focus were it not for examples like Internet applications which require wide-area network (WAN) access when providing observability. Human role multiplicity and non-preemptiveness are the most focused properties. Both can be addressed by dialog functions alone.

The properties thus vary in their architectual specialization and the functional partitions vary in their influence. The relevance of dialog functions is striking. Only device multiplicity and predictability are not related to D (for very different reasons). PI functions have a limited role because many properties are supported by higher-level software processes.

There are no significant differences between the extents of influence of the other three functional partitions. However, FCA functions play a greater role in the provision of interaction flexibility than do FC ones, whereas the reverse is true for the provision of interaction robustness.

4.4 Architecture and Internal Properties

The properties described in this section are properties that are not apparent at the external level. However, the choice of a particular architecture can impact these properties. The influence which the choice of different architectures can have is discussed in the following paragraphs.

Development Efficiency

The existence of an architectural design implies that some thought has been given to software engineering considerations. One of them is efficiency in development of the actual system. Development efficiency is enhanced by a variety of means such as the ability to partition work into manageable pieces. Architectural structures which promote the division of a system's functionality into coherent classes aids in the partitioning and allotment of work.

But equally important are the ways in which these partitions are interconnected. Currently, the systems which most strongly support development efficiency are those which allow a developer to consider a 'vertical slice' of system functionality as one (large) component. Consider, for example, the File Selection widget as it appears in several current systems. Adoption of this widget speeds development not only because it provides

an encapsulation of a commonly used functionality – navigating around the file system and choosing files or directories – but also because it cuts across functional barriers. The File Selection Box contains a Functional Core (the file system), Dialog (when one selects the 'Filter' button, the set of displayed files is updated), and a Presentation (the buttons, sliders, labels and lists which comprise the representation of the widget).

Development efficiency benefits from the ability to *bridge* functional partitions. This, of course, conflicts with separation of concerns, which enhances re-usability – a bridge, or vertical slice, unites what would otherwise be separate functions. Thus if, for example, the developers wanted to re-use a dialog from a previous implementation, then a layered approach would be more appropriate.

To summarize, development efficiency interacts little with the capabilities of specific functional partitions. Instead, it is more impacted by the overall quality of the separation of concerns and the available ways of composing and encapsulating the functional partitions.

Modifiability and Maintainability

The introduction of reference software architectures to the user interface field was motivated by the desire to guarantee modifiability and maintainability of the software. Modifiability and maintainability offer similar design challenges. Modifiability is both supported by, and constrained by, the goals of architecture used. That is, if logically separate functionality is kept physically separate in its architectural realization then the independent modification of those separate functions is supported by this architecture. However, modifications which cut across those functions are not supported by the architecture. The more clearly defined, well motivated, and properly separated architectural components there are, the more modifiable and maintainable the resulting software will be. This separation is achieved through two mechanisms:

- separation of concerns: keeping distinct functional partitions in distinct software components;
- indirection: creating virtual interfaces, such as the logical interaction component, which buffer one component from the implementation details of another component.

To give a concrete example: when modifying a system's dialog the software engineer should not have to worry about the effects of this change on the functional (FC, FCA) or presentation (LI, PI) components, as discussed by Kazman *et al.* (1994). If a new device is added for input or a new representation is needed, one need not change the Dialog component. Furthermore, if one wants to move from one interaction toolkit to another, one should not have to change the Dialog simply because the

attribute names of the interaction objects are slightly different. An architecture which separates these concerns properly supports the modifiability and maintainability of its software.

Thus, *indirection* may be identified as an architectural mechanism that promotes some internal properties. Another architectural mechanism that supports modifiability is the ability to homogeneously decompose a functional partition. In the PAC-Amodeus architecture (Section 4.5.3), the dialog component is refined in terms of a hierarchy of PAC agents.

Portability

Portability is a special case of modifiability. Traditional portability techniques include isolating 'volatile' components into a library, thus localizing the places that may require changes. Portability can thus be supported at an architectural level by separating volatile functional partitions into distinct architectural components. This was one of the main motivations of the Arch/Slinky model.

To give an example, portability across user interface toolkits is simply modifiability with respect to the PI and LI components. Separation of the PI and LI supports portability because under this model the Dialog component of any system ported would be re-usable. By way of contrast, a strict multi-agent approach might have made the porting task much harder because the PI and LI functionality is distributed across the system.

In summary, this internal property is more impacted by pervasive architectural qualities than by a specific functional partitioning. Portability depends on a specialized notion of separation of concerns, such that any volatile functional partitions are isolated from those functions that implement the underlying domain semantics.

Evaluability

This principle is architecture-neutral, since the choice of architecture does not affect how easy it will be to measure the quality of the final system. The concept of an architecture is still important since it may be easier to isolate, in a well-structured system, where evaluation has to be done.

Run Time Efficiency

Given that systems do not have unlimited resources, the efficient usage of system resources is always important. If data have to move through many layers and have to be transformed at each layer then this naturally leads to inefficiency at run time. Thus a layered architecture does not lend itself well to maximizing run time efficiency. As with development efficiency, one needs the ability to take 'vertical slices' of system functionality if high performance is a priority.

For example, it has often been commented that the Seeheim model of user interface software needed to provide a special mechanism to support semantic feedback. This mechanism bridged layers (functional partitions) in the Seeheim model, as indicated by the small, unlabeled box in Figure 4.6, taken from Pfaff (1985).

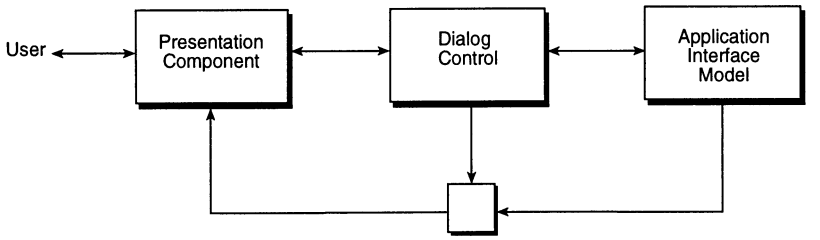

Figure 4.6 *The Seeheim Model.*

Semantic feedback occurs when the presentation is changed in real time according to the *semantics* of a user's input. For example, in the Macintosh desktop, if one drags a file over a folder icon or the dustbin icon, these icons will be highlighted. If, however, one drags a file over another file, the file will not be highlighted. The highlighting of the dustbin or the folder are examples of semantic feedback, indicating that these objects are potential locations for the file being dropped. Semantic feedback involves the use of application (FC) information – knowing the purpose of each screen icon – in the presentation (PI) component.

If one implements each functional partition of the Arch/Slinky model as a separate layer, creating interfaces between each of the five components – FC, FCA, D, LI and PI – then each time the user moves the mouse, an event could pass from LI to FC, and back again.

The frequency of this event may vary. The PI may pass events to the LI each time the mouse moves by a single pixel, but an efficient LI would only pass enter and leave events to D each time the mouse left the bounds of one object and entered another selectable one (the LI can be configured to only pass events for selectable objects, as in PRESENTER as introduced by Took (1990)). Depending on the configurability of the LI, no communication between D and FCA may be required during dragging. With weak abstractions in the LI, and no way for the FCA to indicate selectable objects to the LI (via D), the location information would have to cross eight interfaces. The current pointer position would need to travel from PI to FC (so that the type of the object currently underneath the pointer could be checked) and back again (to actually highlighted the covered object, if necessary). This layering is clearly sub-optimal.

If such advanced LIs and FCAs cannot be easily implemented, one would prefer instead to be able to take a vertical slice of functionality, from FC to

PI, and bundle this functionality together so that as little as possible unnecessary work is done, and as few as possible layer boundaries are crossed.

It sometimes occurs that a system will bridge layers for a small subset of functions, but not for the entire system. This could occur in the following ways.

- An entire Arch could be embedded within a single functional partition. This type of layer bridging is found in any sophisticated widget which provides access to some Functional Core services. An example of this is the File Selection Box widget referred to above.
- A functional partition may directly call a service in another functional partition to which it is not adjacent. An example of this is the layer bridging in the Seeheim model (the 'bypass channel'), as illustrated in Figure 4.6.

In summary, pervasive architectural qualities rather than a specific functional partitioning impact this internal property. Support for 'vertical slice' capabilities as a composition and encapsulation mechanism may improve run time efficiency. But also key architectural decisions on the allocation of function to structure have a direct bearing on this property. The division of labour between LI and PI partitions has a critical effect on the bandwidth of the interfaces between these components. For run time efficiency, we want to minimize the bandwidth requirements among a system's components, particularly if those components are physically separated (say, communicating across a network). The right allocation of function to structure will minimize the events that functions in the D partition must deal with.

Functional Completeness

As defined in Chapter 3, a system is functionally complete if the various abstract commands and state elements required to support the designed task model can be faithfully implemented. Within the scope of this chapter, the term 'faithfully' means that the system is efficient at run time and easy to develop. A good example is the semantic feedback issue discussed above. A strictly layered architecture would here make it difficult to implement this concept faithfully whereas a more fine-grained object oriented architecture like ALV (Hill, 1992) would allow for the necessary trade-offs between maintainability and run time efficiency.

User Interface Integratability

The ability to integrate an application into an existing environment, ensuring that the user interface is compatible with interfaces of other applications in the environment, is a difficult task. It involves achieving a look and feel which is consistent with existing applications. This is typically achieved via user interface toolkits for LI/PI functional partitions. User Interface

Integratability can be achieved through consistent interaction techniques – such as menus, form-filling, drag-and-drop, etc. – and consistent dialogs (e.g. using consistent syntax).

This internal property is the only one with a narrow architectural focus in the presentation components (LI and PI). In part this reflects the rarity of extensible support for semantic external properties such as adaptivity and migratability in current systems.

4.4.1 Internal Properties and Functional Partitions

There are three architectural principles which significantly impact internal properties.

Separation of concerns – or the overall principle of functional partitioning, which varies across the external properties, and brings specific benefits for internal properties where indirection (e.g. virtual interfaces such as LI) is provided.

Division of labor – or the overall allocation of function to structure, which should preserve the overall quality of the functional partitions.

Composition and encapsulation – or the basic provision of structure, which should allow different ways of combining and re-using functional partitions, for example, the 'vertical slices' mentioned above.

For internal properties, the structure of an architecture, and its relation to the functional partitioning rather than the functional partitions themselves, has more profound implications than for external properties. The remainder of this chapter examines these structural issues in more depth.

4.5 Conceptual Architectural Models

The previous two sections introduced the notion of analysing or choosing a systems architecture on the basis of our quality properties. The discussion was based on a generic partitioning of functions for interactive software systems in order to be as broadly applicable as possible.

This and the following section analyse the allocation of function to structure and take a closer look at two conceptual architectural models and two more implementation-oriented software architectures. This aims to reinforce the understanding of the issues introduced above. It also serves to lead up to a concrete example of architectural analysis in Section 4.7.

4.5.1 The Arch Model of Interactive Systems

The Arch model of interactive systems is in UIMS (1992) defined as a layered structure. The functional core of the system to be designed and the UI

toolkit provided by a given implementation environment form its two endpoints between which three additional component layers are interspersed. The model makes direct use of the functional groupings adopted above (with small differences in nomenclature from Figure 4.1). It comprises:

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

Domain-Adapter Component – a mediation component between the Dialog 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-Adapter Component triggers domain-initiated dialog tasks, reorganizes domain data (e.g. collects data items in a list), and detects and reports semantic errors.

Dialog Component – which 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 Dialog and the Interaction Toolkit Components that provides a set of toolkit-independent objects for use by the Dialog 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.

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

Domain objects are used by both the Domain-Specific and the Domain-Adapter 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-Adapter 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 database management system (DBMS) would retrieve a set of employee names and salaries by gender from a database. Iterative review of the list to display parts of succeeding records might need to be done in a Domain-Adapter Component. Here the Domain-Adapter Component would supplement the functionality of the Domain-Specific Component by providing a service related to the presentation of information.

Presentation objects are interaction objects that control user interactions but are toolkit-independent. Presentation objects include descriptions of

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, labeled, 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).

4.5.2 Migration and Branching in the Arch/Slinky metamodel

The above description of Arch could give two misleading impressions:

- functions cannot migrate from their 'logical partition' in an architecture;
- there is no branching into multiple partitions.

Neither is true. Architectural analysis only arises because migration is possible. Otherwise, only one allocation of function to structure would be possible. However, the levels of abstraction given in Chapter 1 do indicate a 'logical partitioning' of functionality in addition to providing important analysis guidance. Too great a departure from the 'spirit' of a functional partition normally reveals itself as a negative impact on internal and/or external properties.

For example, the logical level of interaction requires comprehensive device-independence in its abstraction over physical devices. Development efficiency requires that this is delivered in as compact a form as possible. Also, the dialog level requires clear isolation of *interaction points* in order to support walkthroughs for the assessment of external properties (but especially those determining interaction flexibility). Lastly, the functional level requires capabilities that are compatible with (the user's model of) the work domain. Incompatibility has a negative impact on role multiplicity, predictability, honesty, observability, customizability, and migratability.

In other words: concepts defined at the functional level of abstraction should logically be implemented in the Functional Core (Adapter) component, concepts defined at the dialog level of abstraction should be implemented in the Dialog component, etc. However, the correspondence is not strict; for instance, a strategy of semantic delegation or semantic repair will implement some concepts at the functional level of abstraction in a dialog component, see Bass and Coutaz (1991).

Another migration factor is that sophistication in UIMS's dialog languages is required since dialog functionality (sequencing, constraint maintenance, context management) is inherently complex. Consequently, many

dialog languages are Turing complete, or are extensions of existing Turing complete languages. One can therefore, in principle, do almost anything in the dialog partition of many UIMSs. This does *not*, however, mean that everything in the system is inherently dialog and should be dealt with in one big chunk.

In summary, not only are there good reasons for migration in architectures, but there are many requirements that, once satisfied, make it impossible to prevent migration. Thus when it was stated in Sections 4.3 and 4.4 that a property interacts with some functional partition, this expressed the fact that the property affects the portion of a system which is logically concerned with the partition, irrespective of how or where that functionality is implemented. The analyses thus identify the groups of functions that logically impact properties, leaving aside the problem of what it is *possible to implement.*

Migration is dealt with in the Slinky generalization of the Arch Model as found in UIMS (1992). The coupling of functionalities in the layers of the Arch model described above was designed to minimize the effects of future changes in the interaction toolkit, the user interface dialog 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. However, a model derived to minimize the effects of changing technology may have an adverse effect on the speed of the run time system. A single model cannot satisfy conflicting criteria – i.e. different sets of critical quality properties.

The Slinky metamodel provides a set of Arch models, as opposed to one particular model. The Slinky metaphor was chosen since function groups can migrate through the arch in the way that the coils of a Slinky toy (a large and long spring) may distribute themselves in many different ways throughout their arch.

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. In one sense, the file selection widget can be regarded as a simple string widget at the LI level, but this ignores the extensive file system functions behind it, which can and do make changes to the functional state of the system (such as current drive and directory).

A second complication with architectural models is the need to provide for sub-partitioning of functional classes on the basis of some system context – known as *branching.* This can be added to the Arch/Slinky model as indicated in Figure 4.7 for example.

The causes of branching are varied, but include the following.

Technology – variations in target environments may require multiple adapter partitions, e.g. multiple logical interaction partitions for multiple toolkits at the physical interaction level, or multiple FCA partitions to provide unified access to different kinds of databases (SQL-based, object-oriented, etc.) through the same interface.

Re-use – extensions to a system's capabilities may be achieved by re-use of an existing component. It may be impossible, and will probably be unwise, to incorporate the new component into an existing functional partition.

Closely related applications may need separate Functional Cores, but their interaction with the user should be (almost) identical. One way of handling the addition of separate Functional Cores is to have a single Functional Core Adapter interfacing with all of the Functional Cores. Similarly, for the second factor, a single Logical Interaction component could interface with all Physical Interaction toolkits. However, as long as a Functional Core or Physical Interaction software package doesn't interact with other packages (e.g. by sharing a limited resource like a communication channel, a graphics display device, or a locator device), then it may be preferable to accommodate *multiple* Functional Core Adapters and Logical Interaction components within a system.

Suppose a new Functional Core is added to a system that currently has one instance of each functional partition. Suppose also that the data and functionality of the new Functional Core are independent from that of the old one. In such a case the existing Functional Core Adapter should not be changed to generalize to both Functional Cores, for to do so would ruin the integrity of the existing Functional Core Adapter. In such a case a new Functional Core Adapter should be added to mediate for the new Functional Core, thus isolating the existing Functional Core Adapter from this change to the system as well as from future changes to the new Functional Core. In this way, the two Functional Core Adapters communicate independently with the Dialog Component, forming two branches which join at the Dialog Component (Figure 4.7).

In short, maintaining single instances of functional partitions may not be worth the effort. Branching of the Slinky structure supports multiple instances of functional partitions. Branching also lets a user interface development environment be functionally distributed and modularized by creating networks of components.

These examples of discretionary branching are however less common than imposed branching. In CSCW systems, it is only possible for multiple users to share the same Functional Core (Adapter).* Each user requires a 'leg'

* The delivery of the WYSIWYG (what you see is what you get) property can be controlled by multiple Functional Core Adapters.

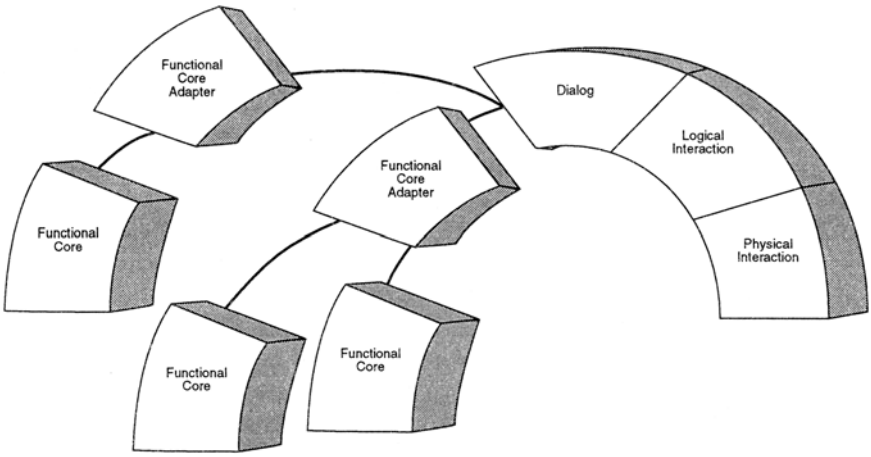

Figure 4.7 *Branching. The Arch/Slinky Metamodel with Multiple FCs and FCAs.*

of the Arch/Slinky branching off from a shared Functional Core (Adapter). This is why branching is the appropriate metaphor for multiple partition instances, since several partitions may be composed. A second example of such branching is when several Functional Cores are visualized on a single screen. This results in several Arch/Slinky structures converging into a single Physical Interaction component.

Branching is also useful for the LI/PI components. Consider an interactive system used by people with different linguistic or cultural background. The same functional core is then equipped with rather different user interfaces implemented as multiple LIs or PIs.

4.5.3 The PAC-Amodeus Conceptual Architectural Model

As alluded to in the previous subsection, branching is a problem not originally addressed in the Arch/Slinky metamodel. Another issue arises when considering recursive decomposition of interactive systems which in many cases is a useful method for designing complex systems, such as systems with highly interactive direct manipulation user interfaces. In such cases layered separation of functionality is not sufficient as the only decomposition principle. The PAC-Amodeus model (Nigay and Coutaz, 1993; Coutaz *et al.*, 1995) is defined as an Arch/Slinky-oriented extension of the original PAC model (Coutaz, 1987) addressing migration, branching and recursive decomposition.

PAC-Amodeus adopts the same components as Arch and assigns the same roles as Arch to these components. However, PAC-Amodeus goes one

step further than Arch by decomposing the Dialog component into a set of cooperative PAC agents, as illustrated in Figure 4.8.

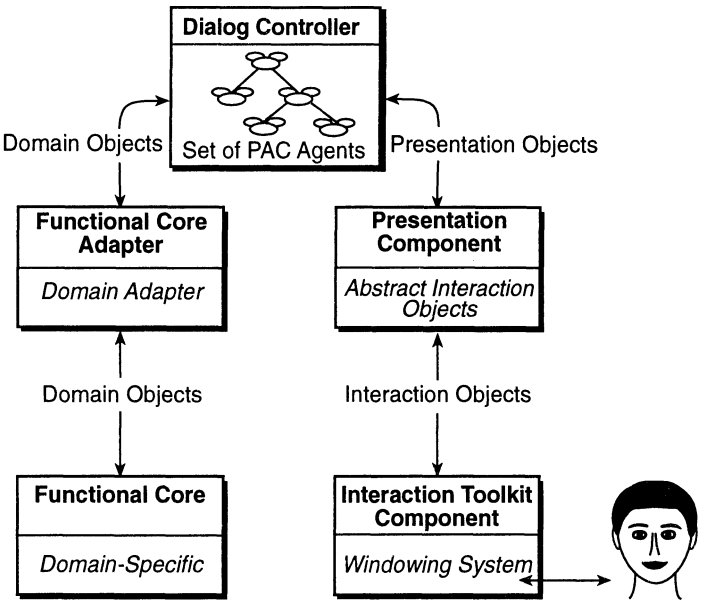

Figure 4.8 *The PAC-Amodeus Model.*

The Dialog Controller has the responsibility for task-level sequencing. Each task or goal of the user corresponds to a thread of dialog. This observation suggests the choice of a multi-agent model which distributes the state of the interaction among a collection of cooperating units. Modularity, parallelism and distribution are convenient mechanisms for supporting multi-thread dialogs. One agent or a collection of cooperating agents can be associated with each thread of the user's activity. Since each agent is able to maintain its own state, it is possible for the user (or the functional core) to suspend and resume any thread at will.

The Dialog Controller receives events both from the Functional Core, via the Functional Core Adapter, and from the user via the Presentation Component. Bridging the gap between a Functional Core Adapter and Presentation Component has some consequences. In addition to task sequencing, the Dialog Controller must perform data transformation and data mapping:

- A Functional Core Adapter and a Presentation Component are oriented in different directions. One is driven by the computational considerations of the Functional Core, the other is toolkit-dependent. In order to match

the two different styles, data must be transformed inside the Dialog Controller.

- State changes in the Functional Core Adapter must be reflected in the Presentation Component (and vice versa). Links must therefore be maintained between domain objects of the Functional Core Adapter and presentation objects in the Presentation Component. A domain object may be rendered with multiple presentation techniques. Therefore, consistency must be maintained between the multiple views of the conceptual object. Such mapping is yet another task of the Dialog Controller.

Thus, bridging the gap between the Functional Core Adapter and the Presentation Component covers task sequencing, formalism translation, and data mapping. Experience shows that these operations must be performed at multiple levels of abstraction and distributed among multiple agents.

Levels of abstraction reflect the successive operations of abstracting and concretion. Abstracting combines and transforms events coming from the presentation techniques into higher-level events for higher abstractions. Conversely, concretion decomposes and transforms high-level data into low-level data. The lowest level of the Dialog Controller is in contact with the presentation objects.

This multi-agent approach supports parallelism, distribution, multithread dialogs and iterative design. Since agents should carry task sequencing, formalism transformation, and data mapping at multiple levels of abstraction, the Dialog Controller is described at multiple grains of resolution combined with multiple facets. At one level of resolution, the Dialog Controller appears as a 'fuzzy potato'. At the next level of description, the main agents of the interaction can be identified. In turn, these agents are recursively refined into simpler agents. This is the usual abstraction/refinement paradigm applied in software engineering.

Orthogonal to this refinement/abstraction axis, the 'facet' axis is introduced. An agent is described along three facets: Presentation, Abstraction, Control. These facets are used to express different but complementary and strongly coupled computational perspectives.

- The Presentation facet of an agent implements the perceivable behavior of the agent. As shown in Figure 4.8, it is related to some presentation object of the Presentation Component.

- The Abstraction implements the competence of the agent (i.e. its expertise) in an essentially media-independent way. It is the Functional Core of the agent. It maintains the abstract state of the agent. It may be related to some domain object(s) of the Functional Core Adapter. The abstraction facet of an agent provides a good mechanism for performing domain-knowledge delegation.

- The Control part of an agent is in charge of two functions: linkage of the Abstraction part of the agent to its Presentation portion and maintenance of the relationships of the agent with other agents. The linkage serves two purposes: i) formalism transformations between the Abstraction and the Presentation portions of the agent, and ii) data mapping between the abstract facet and the presentation facet. Relationships between agents may be static or dynamic. Dynamic relationships are required when agents are dynamically created/deleted. Relationship maintenance by the control part of an agent covers the communication and the synchronization mechanism between this agent and its cooperating partners.

In summary, a PAC agent could be viewed as a mini-Arch. Figure 4.9 shows how one PAC agent relates to other agents and to the surrounding world of the Dialog Controller:

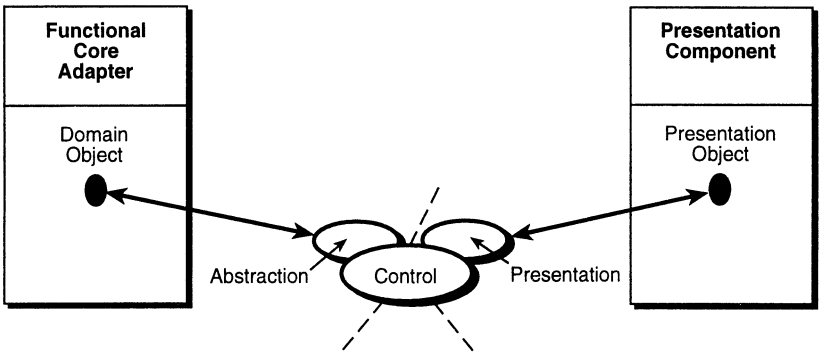

Figure 4.9 *A PAC agent of the Dialog Controller.*

4.6 Example Architectures

In the following examples the Serpent and Chiron-1 architectures are discussed in the light of the generic functional partitioning described above and from the viewpoint of satisfying the quality properties introduced. Research systems have been chosen for the following reasons.

- Research tools are often concerned with software architecture, and so provide enough information in their descriptions to allow architectural analysis. This is often not the case for commercial systems.
- One commercial tool should not be advocated over another in this context.

A graphical convention for representing architectural structure is introduced, see Figure 4.10. This in combination with the generic functional partitioning will ease comparisons and discussions of the architectures.

Rectangles with solid lines represent processes, or independent threads of control, ovals represent computational components which only exist within a process or within another computational component (e.g. procedures or modules), shaded rectangles represent passive data repositories (typically files), shaded ovals represent active data repositories (e.g. active databases), solid arrows represent data flow (uni- or bi-directional) and grey arrows represent control flow (also uni- or bi-directional).

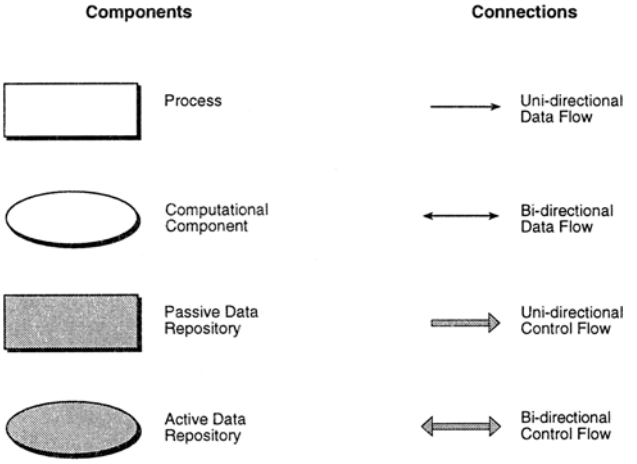

Figure 4.10 *Structural Notations.*

4.6.1 Serpent

Serpent identifies a dialog controller, the presentation and the application as three distinct processes in its architecture, as shown in Figure 4.11.

Application modules contain the computational semantics required for the application. Although there can theoretically be many different applications contained within a given run time instance of Serpent, there is typically only one. Presentation modules provide techniques for supporting interaction at both the logical and physical level completely independently of application semantics. Different presentation modules in a given run time instance are possible, although not typical. Given that application and presentation modules are separate, there must be a way to coordinate a given application component with a presentation component. That is the purpose of the dialog controller. The dialog component mediates the user's interaction with an application, through the control of the presentation.

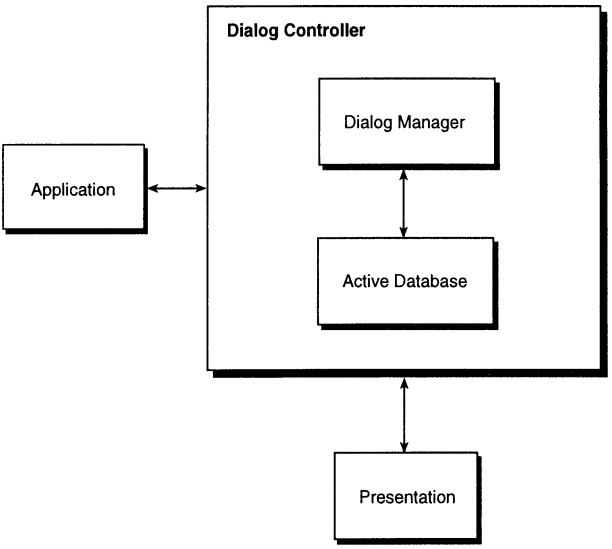

Figure 4.11 *Serpent's Architecture.*

All communication between Serpent components is mediated by constraints on shared data in the database shown in Figure 4.11. This structure is implemented as an active database; when values in the database change, they are automatically communicated to any component which is registered as being interested in the data. This global database physically resides in the same process as the dialog controller but is logically independent from all of the Serpent components. A dialog manager sits within the dialog controller process and mediates the connection between application and presentation. The dialog manager is further decomposed into a collection of view controllers – not shown in Figure 4.11 – which provide a finer grain of correspondence between application and presentation objects.

4.6.2 Analysis of Serpent

This section explains the mapping between the system-specific notion which Serpent implements and the functional partitioning used as the base of reference in this book. Figure 4.12 recasts Serpent's architecture in the common functional notation given in Section 4.2.

Several things have been changed between Figure 4.11 and Figure 4.12:

- the Active Database is represented as an active repository;
- data and control relationships are exposed;
- independent flows of control are exposed through the delineation of processes; and

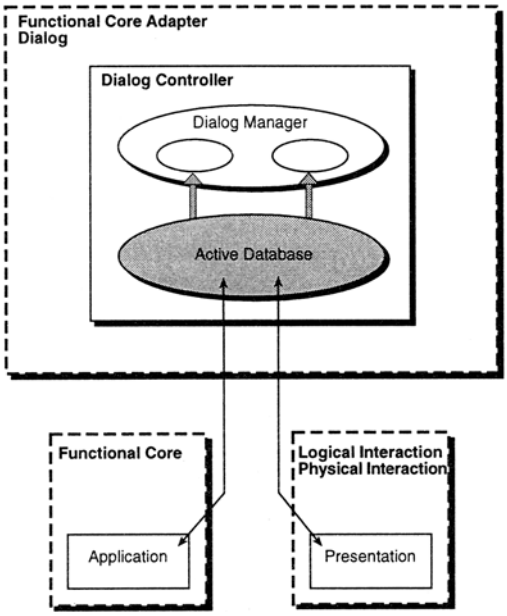

Figure 4.12 *Serpent's Architecture (annotated).*

- a view controller hierarchy is exposed as a subdivision of the dialog manager.

Recall that, with properties associated with functional roles, and with Serpent analysed in terms of these roles, Serpent can now be assessed against a property profile. This assessment will be discussed after the analysis of another example architecture.

4.6.3 Chiron-1

Chiron-1 (Taylor and Johnson, 1993) is a User Interface Design System which was created with the goal of addressing two important software life-cycle issues: maintainability and sensitivity to environmental changes. Chiron-1's architecture, as presented by its authors, is shown in Figure 4.13.

A Chiron-1 system consists of a client and a server. The client consists of an application, which exports a number of abstract data types (ADTs) which Chiron-1 encapsulates within Dispatchers. Dispatchers communicate with Artists, which maintain abstract representations of their associated ADTs in terms of an abstract depiction library (ADL).

A Chiron-1 server consists of: a virtual window system, which translates from abstract interface depictions into concrete ones; and an instruc-

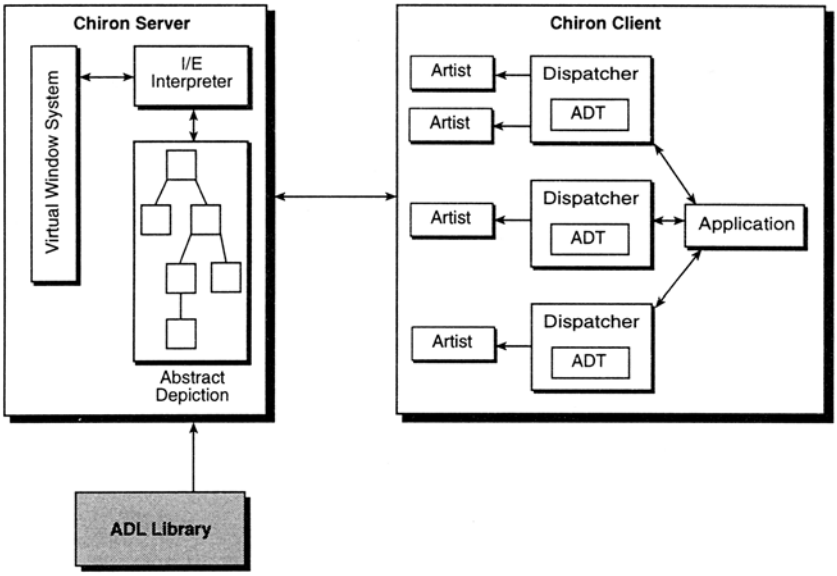

Figure 4.13 *Chiron-1's Architecture (original).*

tion/event interpreter. It also accesses an ADL. The instruction/event interpreter responds to requests from Artists to change the abstract description and translates those requests into changes to the presentation. The server also responds to events from users and translates those back into Artist requests.

All of these components – ADL, Artists, virtual window system etc. – are specific to Chiron-1. This 'naming problem' makes analysis and comparison with other systems difficult. The goal in doing architectural analysis is to have a single language and a single representation for understanding architectural issues. In order to do this, Chiron-1 is re-characterized in terms of the functional roles given in Section 4.2.1. This leads to a system-independent language for talking about functionality, and the partitioning of functionality.

4.6.4 Analysis of Chiron-1

The Chiron-1 architecture clearly separates the application (functional core) from the rest of the system, as would be expected in a system which was built with the expressed goal of minimizing sensitivity to environmental changes. The functional core adapter could live in the ADTs, or in the Artists. It seems clear that the Artists contain some of the dialog since

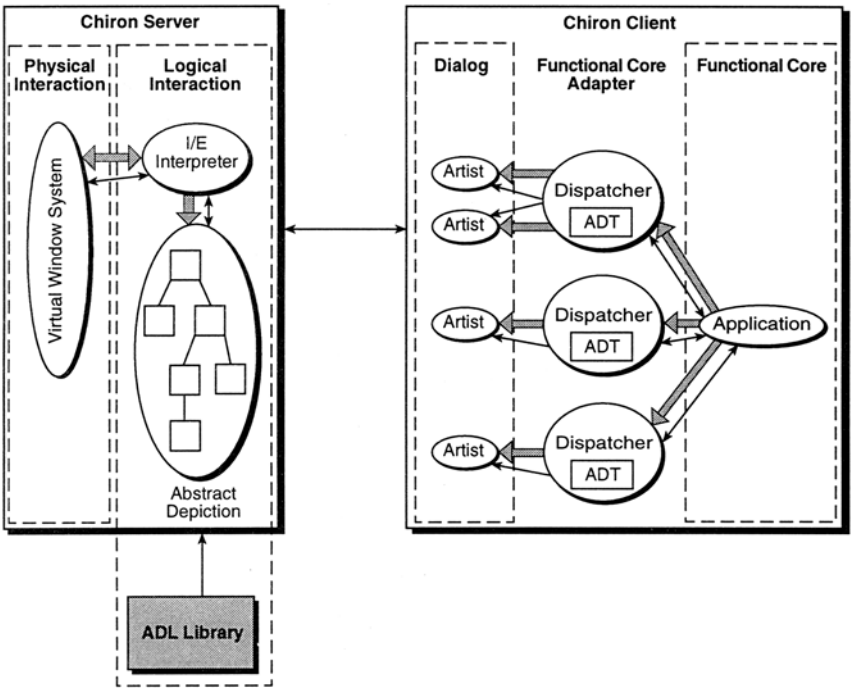

Figure 4.14 *Chiron-1's Architecture (annotated).*

they have the job of maintaining a correspondence between objects from the application domain and interface objects from the presentation domain.

However, what is less clear from Chiron-1's architectural description is where the 'state' of the dialog lives. For example, where does one put the information that the 'Paste' option in an edit menu should be greyed out unless something has previously been cut or copied? Another type of dialog issue is maintaining relationships among the interface objects. For example, when a user selects the 'Save As' option in a file menu, something in the dialog must cause a file selection box to be created.

The location of these sorts of dialog issues is explicitly addressed in Chiron-1's architectural description. These dependencies might exist in the Artists, in the ADTs or even in the Abstract Depiction Libraries. For simplicity's sake, we have provisionally annotated the architecture to show the dialog as living completely in the Artists.

The final two functional roles are clearly identified: the Physical Interaction function corresponds to Chiron-1's Virtual Window System component, and the Logical Interaction functionality is provided by the I/E

interpreter, augmented by the ADL. As a result of this characterization, the Chiron-1 architecture is provisionally annotated as shown in Figure 4.14.

In this depiction of Chiron-1 (which was adapted from the original system architecture given in Taylor and Johnson (1993)), the partitioning of functionality can be viewed in terms of the functional roles given in Section 4.2.1.

4.7 Assessing Quality Properties

The main aim in analysing the allocation of functional roles to a specific architectural structure is to support assessment of an architecture against a property profile (as long as properties have been associated with functional roles).

Before giving examples of such an assessment for both Serpent and Chiron-1, the above analysis of Chiron-1 can be cross-checked by mapping the temperature/humidity example onto Chiron-1's architecture. Performing this mapping has the further advantage of allowing assessment of properties for a specific realization of an architecture for a specific (small) computer system.

4.7.1 A Chiron-1 Architecture for a Climate Control System

The functional core of the climate control system (Section 4.2.2) – getting current or ambient temperature and humidity and setting the desired temperature and humidity – will be located in the 'Application' component, labeled FC in Figure 4.14. The functional core adapter – responsible for conversion and bundling/unbundling of information – will be located in the Chiron-1 'ADT Dispatcher' components, labeled FCA.

The dialog functionality for the climate control system – reporting user requests to the application, displaying application information to the user and switching between setting mode and ambient temperature mode – should be located in the Artists, as indicated by the D label surrounding the Artists.

The presentation functionality in Chiron-1 is all located in the Chiron-1 server. The logical interaction portion, which translates between generic presentation objects and particular window-system specific objects, is located in Chiron-1's Abstract Depictions and I/E interpreter, as indicated by the LI label. The physical interaction component, which would display the controller-gauge widgets to the user and receive input from the user, is located in Chiron-1's Virtual Window System, as indicated by the PI label.

The functions of the climate control system are now partitioned and mapped onto the structural components of a realized user interface architecture. This allows establishment of the software ramifications of user-

oriented properties. If it were desirable, for example, to guarantee a particular property of a user interface – e.g. that a user can always return to a previous state no matter what their current state is – it would be necessary to assess how this property would affect the software of the system being developed. One key concern is to know which portions of the system need to be changed to better support a property. This is done for a few sample properties below.

4.7.2 Assessment of Properties

Representation Multiplicity

Consider, for example, the property of representation multiplicity. As stated in Section 4.3.2, representation multiplicity can be manifested in a number of different ways – it does not necessarily belong to any one function or one software component. Two possibilities suggested were:

- the FC could pass a single object to the FCA, which would then split it into two objects for the D;
- a single application object could be passed from the FC to the D, and from the D converted into two LI objects.

Consider how this would be manifested in a system. A system such as Chiron-1 could implement the first possibility. A single application object, created in the FC, would then be packaged by Chiron-1 as an ADT (in the FCA). This ADT would be split into two realizations and passed, via a Dispatcher to two distinct Artists (dialog components).

In another system, one might choose to split an application object at a different point. For example, the Serpent User Interface Management System described in Bass *et al.* (1990) divides a system into FC, D/FCA and PI/LI components, as shown in Figure 4.12.

In Serpent, the FCA and D functionality are undifferentiated, and so there is only one way to achieve representation multiplicity: a single application object is passed to the FCA/D component, where it is split into two. Each of these two objects in Serpent's Active Database would then correspond to distinct LI/PI objects.

Insistence

The same kind of analysis of a software architecture can be made with respect to other properties as, for instance, the property of insistence (Section 4.3.3). Insistence, it should be remembered, deals with the duration and period of the effect of a communication act. Consider a visual interaction object which is to have a have a blinking aspect.

In order to achieve this function in Chiron-1, the designer has a choice between three possibilities. If the underlying medium provides this function,

it will be located in the PI – Chiron-1's Virtual Window System – and he or she needs only choose the appropriate objects and attributes. If the PI does not provide it, one would ideally want to simulate this functionality in the LI. The LI provides a consistent set of interaction objects to the Dialog, smoothing over or hiding the idiosyncratic differences of individual toolkits.

In order to achieve insistence then, the LI would have to simulate an insistent visual object by 'blinking' it – alternating its background and foreground colors. In the Chiron-1 system, this functionality would have to be located in the Abstract Depiction library. If, however, the LI could not or did not support such functionality, then it would have to be supported in the dialog component – in the Artists, in the case of Chiron-1.

In Serpent roughly the same situation is found. Insistence could be supported in Serpent's Dialog Controller, or in its Presentation (which does not architecturally differentiate Logical Interaction and Physical Interaction).

Modifiability

Modifications to the user interface are easiest to perform when the fewest modules must be changed. Let us examine one specific modification: 'grey out menu items that are currently not accessible'. This modification requires knowledge of currently valid menu choices and this knowledge is contained in the Dialog. Both Serpent and Chiron-1 isolate Dialog within their architectures and so this type of modification is assisted by both of them.

User Interface Integratability

Suppose that a new system written using Serpent or Chiron-1 is to be introduced into an existing environment. Since the details of the presentation come from the Physical Interaction partition, and since this partition is localized in Chiron-1 it can be seen that this architecture assists User Interface Integratability. Since in Serpent the Logical Interaction and the Physical Interaction partitions are bundled together, Serpent is less effective in this than Chiron-1.

4.8 Conclusion

Chapter 3 introduced a set of software phenomena that interacted with properties under the broad heading of software techniques. It identified software architecture as the phenomenom that interacted most with our properties. In this chapter, we have examined these interactions. To do this, we have described software architectures in more detail, and have presented a common framework for architectural analysis. The functional groupings that were adopted in this framework can be related to our prop-

erties and thus identify where attention can be focused when attempting to satisfy a property. This helps with the choice of an architectural model that is optimal for a prioritized list of required properties. Even so, choice between different published software architectures is difficult. There are always different ways of applying a given architectural model to a given design problem, especially as their descriptions in the literature are not sufficiently detailed. Architectural analysis would become more straightforward if architectural models were accompanied by guidelines on how to apply them. Furthermore, they could be further motivated with reference to those properties that they supported well.

Despite these difficulties, we have shown how properties can guide the choice of a system software architecture. Such guidance is most straightforward when properties are associated with a single functional grouping. For example, reachability and properties that involve mappings by dialog functions such as observability are most easily checked for in architectures such as Serpent where the dialog is explicitly represented and localized. Analysis of these properties is far less straightforward for architectures such as PAC-Amodeus or Chiron-1 where the dialog state is distributed across multiple agents or artists. Conversely, multi-threading, representation multiplicity and device multiplicity are more easily handled within multi-agent approaches like PAC-Amodeus where each thread of dialog is realized by a separate sub-hierarchy of agents. For both sets of examples, analysis is simpified by our prior association of properties with functional groupings. Once an architecture has been related to these functional groupings, a property profile can easily be established for an architectural model, which in turn simplifies the analysis of specific instances of a software architecture.