Technical Report
CMU/SEI-90-TR-22
ESD-90-TR-223

A Design Space and Design Rules for User Interface Software Architecture

Thomas G. Lane

November 1990

Technical Report

CMU/SEI-90-TR-22
ESD-90-TR-223
November 1990

A Design Space and Design Rules for User Interface Software Architecture

Thomas G. Lane

School of Computer Science
Software Architecture Design Principles Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

A Design Space and Design Rules for User Interface Software Architecture

Abstract. The architecture of a user interface software system can be described in terms of a fairly small number of key functional and structural choices. This report presents a "design space" that identifies these key choices and classifies the alternatives available for each choice. The design space is a useful framework for organizing and applying design knowledge. The report presents a set of design rules expressed in the terms of the design space. These rules can help a software designer to make good structural choices based on the functional requirements for a user interface system. Extension of this work might eventually provide automated assistance for structural design.

1. Introduction

Software architecture is the study of the large-scale structure and performance of software systems [Shaw 89]. Important aspects of a system's architecture include the division of functions among system modules, the means of communication between modules, and the representation of shared information. This report describes the architecture of user interface systems, using a *design space* that identifies the key architectural choices and classifies the available alternatives. The space provides a framework for *design rules* that can assist a designer in choosing an architecture that is appropriate for the functional requirements of a new system. The design space is useful in its own right as a shared vocabulary for describing and understanding systems.

This report is a summary of results from the author's thesis [Lane 90a]. It concentrates on presenting those results that are of interest to user interface system builders. A companion report argues that design spaces and rules may be a widely applicable means of expressing software engineering knowledge [Lane 90b].

1.1. Rationale

The established fields of engineering have long distinguished between routine and innovative design methods. *Routine design* uses standardized methods to solve problems similar to those that have been solved before. This process is not expected to yield the best possible design, but rather to yield a design that meets all the stated requirements with minimum design effort. In contrast, *innovative design* methods rely less on prior practice than on raw invention or derivation from abstract principles. Innovative designs can solve new types of problems or produce solutions especially well-tuned to specific requirements, but at a high design cost. Moreover, innovative design is more likely to fail to produce a solution than routine design (where a routine method is applicable). Engineering handbooks (e.g., [Perry 84]) exist primarily to support routine design. A good handbook arms its user with a number of standard design approaches and with knowledge of their strengths and limitations.

Routine design methods have benefits beyond reducing initial design cost. A standardized, commonly known design method reduces the effort needed to understand another person's design; hence, maintenance costs are also reduced. More fundamentally, standardized methods provide a context for the creation and application of knowledge; this is why a standardized method is usually better understood and more reliable than an ad hoc one. For example, the recognition and use of standard control flow patterns (conditionals, iteration, and so forth) made it possible for researchers to discover the key properties of those patterns (e.g., invariant and termination conditions of loops). Programmers now routinely use this knowledge to produce better-quality code than was possible without it.

At present, routine design is not well practiced by software engineers. Some designers tend to invent every system from scratch, while others tend to reuse a familiar design regardless of its suitability. Both errors arise from lack of a set of standardized methods. Handbook-like texts are now widely available for selection of algorithms and data structures (e.g., [Knuth 73, Sedgewick 88]), but such handbooks do not yet exist for higher levels of software design. The work reported here is a start toward developing a routine practice of software system architecture, within the limited domain of user interface systems.

The systems covered by this study are those whose main focus is on providing an interactive user interface for some software function(s). This includes user interface management systems (UIMSs), graphics packages, user interface toolkits, window managers, and even stand-alone applications that have a large user interface component. This range is large enough that no single design can cover all cases; hence, we must consider how to choose among alternatives. At the same time, the range is not too large to allow recognition of common patterns. Future work may make it possible to construct useful design spaces for larger classes of software systems.

1.2. The Notion of a Design Space

The central concept in this report is that of a multi-dimensional design space that classifies system architectures. Each dimension of a design space describes variation in one system characteristic or design choice. Values along a dimension correspond to alternative requirements or design choices. For example, required response time could be a dimension; so could the means of interprocess synchronization (e.g., messages or semaphores). A specific system design corresponds to a point in the design space, identified by the dimensional values that correspond to its characteristics and structure. Figure 1-1 illustrates a tiny design space.

A design dimension is not necessarily a continuous scale; in most cases the space considers only a few discrete alternatives. For example, methods for specifying user interface behavior include state transition diagrams, context-free grammars, menu trees, and many others. Each of these techniques has many small variations, so one of the key problems in constructing a design space is finding the most useful granularity of classification. Even when a dimension is in principle continuous, one may choose to aggregate it into a few

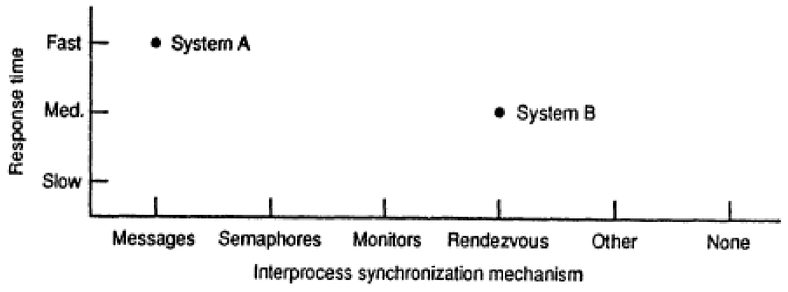

Figure 1-1: A Simple Design Space

discrete values (e.g., "low," "medium," "high"). This is appropriate when such gross estimates provide as much information as one needs (or can get) in the early stages of design.

Another way in which a design space differs from geometric intuition is that the dimensions may not be independent. In fact, it is important to discover correlations between the dimensions in order to create design rules describing appropriate and inappropriate combinations of choices. One empirical way of discovering such correlations is to see whether successful system designs cluster in some parts of the space and are absent from others.

A crucial part of this approach is to choose some dimensions that reflect requirements or evaluation criteria (function and/or performance), as well as other dimensions that reflect structure (or other available design choices). Then, the observed correlations and resulting design rules can provide direct design guidance: they show which design choices are most likely to meet the functional requirements for a new system.

1.3. Related Work

The concept of a design space is far from new. A seminal use is Bell and Newell's taxonomy of computer hardware structures [Bell 71]. They describe computers using dimensions such as function (e.g., numeric calculation or communication), instructions per second, memory size, and hardware-supported data types. A software-oriented example is Wegner's design space for object-oriented languages [Wegner 87].

The domain covered by this report's design space is user interface software. Various researchers have investigated individual aspects of user interface software structures. Most prior work deals with control flow patterns [Hayes 85, Tanner 83] or classification of notations for user interface appearance and behavior [Green 86, Myers 89]. Other workers have made proposals for standard module structures [Dance 87, Lantz 87]. Hartson and Hix survey much of the existing work [Hartson 89]. For the most part, however, the user interface research community has neglected internal structural issues in favor of work on selection and description of the external behavior of a user interface. Hence the work reported here provides a more complete view of the space of user interface structural alternatives than any prior work and, for several of the previously investigated dimensions, it offers new classifications that are more useful for making structural decisions.

2. An Overview of the Design Space

This section introduces the design space by describing a dozen of its dimensions. The complete space includes nearly fifty dimensions, many of which are fairly obvious to anyone familiar with user interface software. To avoid bogging down in details, we will consider only the more interesting dimensions. For a complete description of the space, see Appendix A.

2.1. A Basic Structural Model

To describe structural alternatives, it is necessary to have some terminology that identifies components of a system. The terminology must be quite general, or it will be inapplicable to some structures. A useful scheme for user interface systems divides any complete system into three components, or groups of modules:

1. An **application-specific** component: Code that is specific to one particular application program and is not intended to be reused in other applications. In particular, this component includes the functional core of the application. It may also include application-specific user interface code. (The term "code" should be read as including tables, grammars, and other non-procedural specifications, as well as conventional programming methods.)
2. A **shared user interface** component: Code that is intended to support the user interface of multiple application programs. If the software system can accommodate different types of I/O devices, only code that is applicable to all device types is included here.
3. A **device-dependent** component: Code that is specific to a particular I/O device class (and is not application-specific).

In a simple system, the second or third component might be empty: there might be no shared code other than device drivers, or the system might have no provision for supporting multiple device types (and hence no clear demarcation of device-specific code).

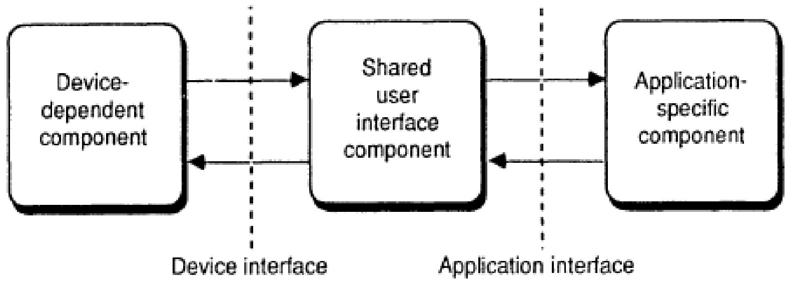
Device-dependent component
Shared user interface component
Application-specific component
Device interface
Application interface

Figure 2-1: A Basic Structural Model for User Interface Software

The intermodule divisions that the design space considers are the division between application-specific code and shared user interface code on the one hand, and between device-specific code and shared user interface code on the other. These divisions are called the *application interface* and *device interface* respectively. Figure 2-1 illustrates the structural model.

There is some flexibility in dividing a real system into these three components. This apparent ambiguity is very useful, for one can analyze different levels of the system by adopting different labelings. For example, in the X Window System [Scheifler 86], one may analyze the window server's design by regarding everything outside the server as application specific, then dividing the server into shared user interface and device-dependent levels. To analyze an X toolkit package, it is more useful to label the toolkit as the shared code, while regarding the server as a device-specific black box.

2.2. Functional Design Dimensions

The functional dimensions identify the user interface system requirements that most affect the system's structure. These dimensions are not intended to correspond to the earliest requirements that one might write for a system, but rather to identify the specifications that immediately precede the gross structural design phase. Thus, some design decisions have already been made in arriving at these choices.

The first example of a functional dimension is **command execution time**. This dimension indicates how long the application program may take to process a command, compared with the reaction time of a human user. Useful classifications are:

- **Short maximum time:** All commands can be executed in a short time, say a few tenths of a second.
- **Intermediate maximum, short average:** Most commands are executed in a short time, but some may take a bit longer, up to a couple of seconds.
- **Long maximum time:** Some or all commands may take a long time to execute so that the user will have a strong perception of waiting.

As an example of the importance of command execution time, a system in the first category can probably dispense with handling asynchronous input (i.e., no type-ahead or command cancellation features). This is less likely to be appropriate when long-running commands are present.

The second example functional dimension is **external event handling**: does the application program need to respond to external events, that is, events not originating in the user interface? If so, on what time scale?

- **No external events:** The application is uninfluenced by external events, or checks for them only as part of executing specific user commands. For example, a mail program might check for new mail, but only when an explicit command to do so is given. In this case, no support for external events is needed in the user interface.
- **Process events while waiting for input:** The application must handle exter-

nal events, but response time requirements are not so stringent that it must interrupt processing of user commands. It is sufficient for the user interface to allow response to external events while waiting for input.

- **External events preempt user commands:** External event servicing has sufficiently high priority that user command execution must be interrupted when an external event occurs.

Like the previous dimension, external event handling has obvious implications for control flow within the user interface and application.

The third example of a functional dimension is **user interface adaptability across devices**. This dimension measures how much change in user interface behavior may be required when changing to a different set of I/O devices:

- **None:** All aspects of behavior are the same across all supported devices. (This includes the case that only one set of I/O devices is supported.)
- **Local behavior changes:** Only changes in small details of behavior across devices; for example, the appearance of menus.
- **Global behavior changes:** Major changes in surface user interface behavior; for example, a change between menu-driven and command-language interface types.
- **Application semantics changes:** Changes in underlying semantics of commands (e.g., continuous display of state versus display on command).

The final examples are a complementary pair of dimensions. **Application portability across interaction styles** specifies the degree of portability across interaction styles required for applications that will use the user interface software:

- **High:** Applications should be portable across significantly different styles (e.g., command language versus menu-driven).
- **Medium:** Applications should be independent of minor stylistic variations (e.g., menu appearance).
- **Low:** User interface variability is not a concern, or application changes are acceptable when modifying the user interface.

User interface system adaptability across interaction styles specifies how adaptable to different interaction styles the shared user interface software should be:

- **High:** Adaptable to a wide range of interface styles.
- **Medium:** Limited adaptability.
- **Low:** Imposes a specific interface style.

Since interface behavior must be specified somewhere, there is a tradeoff between application and shared user interface flexibility: either the shared software imposes a stylistic decision, or the application makes the decision and hence becomes less portable. This dilemma can be alleviated by wise use of default choices, but in general, high requirements for both of these dimensions should be viewed with suspicion. In the other direction, low requirements for both dimensions indicate little flexibility in user interface behavior, which is perfectly appropriate for some systems (for example, if strong user interface conventions exist).

2.2.1. The Most Important Functional Dimensions

It is reasonable to expect that some functional dimensions have more influence on structure than others, but it is difficult to guess which ones have the greatest impact. Some insight can be gained from the author's experiments with automated design rules (see Section 4): we can rank the functional dimensions according to the total weight given to each in the automated rule set. (Those rules did not fully reproduce the decisions of human experts, so this ranking may need to be modified when better data is available.) On this basis, the five functional dimensions with most influence on the structural dimensions are:

- User interface system adaptability across devices
- Application portability across devices
- Application portability across interaction styles
- Basic interface class
- System organization

The next five dimensions are:

- Available processing power
- I/O device class breadth
- User interface system adaptability across interaction styles
- User customizability
- External event handling

The remaining fifteen functional dimensions (listed in Appendix A) have less influence on structure.

The most striking feature of this ranking is the importance of dimensions having to do with flexibility. Evidently the nature and degree of adaptability required of the system are by far the most important determinants of an appropriate structure. It is an open question whether this property is unique to user interface system structures, or is true for other kinds of software as well.

2.3. Structural Design Dimensions

This section presents some important structural dimensions: the fundamental decisions about system structure.

Application interface abstraction level is in many ways the key structural dimension. The design space identifies six general classes of application interface, which are most easily distinguished by the level of abstraction in communication:[1]

- **Monolithic program:** There is no separation between application-specific and shared code, hence no application interface (and no device interface, either). This can be an appropriate solution in small, specialized systems where the application needs considerable control over user interface details and/or little processing power is available. (Video games are a typical example.)
- **Abstract device:** The shared code is simply a device driver, presenting an abstract device for manipulation by the application. The operations provided have specific physical interpretations (e.g., "draw line," but not "present menu"). Most aspects of interactive behavior are under the control of the application, although some local interactions may be handled by the shared code (e.g., character echoing and backspace handling in a keyboard/display driver). In this category, the application interface and device interface are the same.
- **Toolkit:** The shared code provides a library of interaction techniques (e.g., menu or scroll bar handlers). The application is responsible for selecting appropriate toolkit elements and composing them into a complete interface; hence the shared code can control only local aspects of user interface style, with global behavior remaining under application control. The interaction between application and shared code is in terms of specific interactive techniques (e.g., "obtain menu selection"). The application can bypass the toolkit, reaching down to an underlying abstract device level, if it requires an interaction technique not provided by the toolkit. In particular, conversions between specialized application data types and their device-oriented representations are done by the application, accessing the underlying abstract device directly.[2]
- **Interaction manager with fixed data types:** The shared code controls both local and global interaction sequences and stylistic decisions. Its interaction with the application is expressed in terms of abstract information transfers, such as "get command" or "present result" (notice that no particular external representation is implied). These abstract transfers use a fixed set of standard data types (e.g., integers, strings); the application must express its input and output in terms of the standard data types. Hence some aspects of the conversion between application internal data formats and user-visible representations remain in the application code.

[1]Recognition of abstraction level as a key property in user interfaces goes back at least to Hayes et al [Hayes 85]. The classification used here is a practical one, but it is based on the theoretical distinctions made by Hayes.

[2]The notion that conversion between internal and external representations of data types is a key activity in user interfaces is due to Shaw [Shaw 86].

- **Interaction manager with extensible data types:** As above, but the set of data types used for abstract communication can be extended. The application does so by specifying (in some notation) the input and output conversions required for the new data types. If properly used, this approach allows knowledge of the external representation to be separated from the main body of the application.
- **Extensible interaction manager:** Communication between the application and shared code is again in terms of abstract information transfers. The interaction manager provides extensive opportunities for application-specific customization. This is accomplished by supplying code that augments or overrides selected internal operations of the interaction manager. (Most existing systems of this class are coded in an object-oriented language, and the language's inheritance mechanism is used to control customization.) Usually a significant body of application-specific code customizes the interaction manager; this code is much more tightly coupled to the internal details of the interaction manager than is the case with clients of nonextensible interaction managers.

This classification turns out to be sufficient to predict most aspects of the application interface, including the division of user interface functions, the type and extent of application knowledge made available to the shared user interface code, and the kinds of data types used in communication. For instance, we have already suggested the division of local versus global control of interactive behavior that is typically found in each category.

Abstract device variability is the key dimension describing the device interface. We view the device interface as defining an *abstract device* for the device-independent code to manipulate. The design space classifies abstract devices according to the degree of variability perceived by the device-independent code:

- **Ideal device:** The provided operations and their results are well specified in terms of an "ideal" device; the real device is expected to approximate the ideal behavior fairly closely. (An example is the PostScript imaging model, which ignores the limited resolution of real printers and displays [Adobe 85].) In this approach, all questions of device variability are hidden from software above the device driver level, so application portability is high. This approach is most useful where the real devices deviate only slightly from the ideal model, or at least not in ways that require rethinking of user interface behavior.
- **Parameterized device:** A class of devices is covered, differing in specified parameters such as screen size, number of colors, number of mouse buttons, etc. The device-independent code can inquire about the parameter values for the particular device at hand, and adapt its behavior as necessary. Operations and their results are well specified, but depend on parameter values. (An example is the X Windows graphics model, which exposes display resolution and color handling [Scheifler 86].) The advantage of this approach is that higher level code has both more knowledge of acceptable tradeoffs and more flexibility in changing its behavior than is possible for a device driver. The drawback is that device-independent code may have to perform complex case analysis in order to handle the full range of supported devices. If this must be done in each application, the cost is high and there is a great risk that programmers will omit support for some devices. To reduce this temptation, it is best to design a

parameterized model to have just a few well-defined levels of capability, so as to reduce the number of cases to be considered.

- **Device with variable operations:** A well-defined set of device operations exists, but the device-dependent code has considerable leeway in choosing how to implement the operations; device-independent code is discouraged from being closely concerned with the exact external behavior. Results of operations are thus not well specified. (For example, GKS logical input devices [Rosenthal 82] and the Scribe formatting model [Reid 80].) This approach works best when the device operations are chosen at a level of abstraction high enough to give the device driver considerable freedom of choice. Hence the device-independent code must be willing to give up much control of user interface details. This restriction means that direct manipulation (with its heavy dependence on semantically-controlled feedback) is not well supported.
- **Ad-hoc device:** In many real systems, the abstract device definition has developed in an ad-hoc fashion, and so it is not tightly specified; behavior varies from device to device. Applications therefore must confine themselves to a rather small set of device semantics if they wish to achieve portability, even though any particular implementation of the abstract device may provide many additional features. (Alphanumeric terminals are an excellent example.) While aesthetically displeasing, this approach has one redeeming benefit: applications that do not care about portability are not hindered from exploiting the full capabilities of a particular real device.

These categories lend themselves to different situations. For example, abstract devices with variable operations are useful when much of the system's "intelligence" is to be put into the device-specific layer; but they are only appropriate for handling local changes in user interface behavior across devices.

Notation for user interface definition classifies the techniques used for defining user interface appearance and behavior:

- **Implicit in shared user interface code:** Information "wired into" shared code. For example, the visual appearance of a menu might be implicit in the menu routines supplied by a toolkit. In systems where strong user interface conventions exist, this is a perfectly acceptable approach.
- **Implicit in application code:** Information buried in the application and not readily available to shared user interface code. This is most appropriate where the application is already tightly involved in the user interface, for example, in handling semantic feedback in direct manipulation systems.
- **External declarative notation:** A nonprocedural specification separate from the body of the application program, for example, a grammar or tabular specification. External declarative notations are particularly well suited to supporting user customization and to use by nonprogramming user interface experts. **Graphical specification** methods are an important special case.
- **External procedural notation:** A procedural specification separate from the body of the application program; often cast in a specialized programming language. Procedural notations are more flexible than declarative ones, but are harder to use. User-accessible procedural mechanisms, such as macro defini-

tion capability or the programming language of EMACS-like editors [Borenstein 88], provide very powerful customization possibilities for sophisticated users. However, an external notation by definition has limited access to the state of the application program, which may restrict its capability.

- **Internal declarative notation:** A nonprocedural specification within the application program. This differs from an implicit representation in that it is available for use by the shared user interface code. Parameters supplied to shared user interface routines often amount to an internal declarative notation. An example is a list of menu entries provided to a toolkit menu routine.
- **Internal procedural notation:** A procedural specification within the application program. This differs from an implicit representation in that it is available for use by the shared user interface code. A typical example is a status-inquiry or data transformation function that is provided for the user interface code to call. This is the most commonly used notation for customization of extensible interaction managers. It provides an efficient and flexible notation, but is not accessible to the end user, and so is useless for user customization. It is particularly useful for handling application-specific feedback in direct manipulation interfaces, since it has both adequate flexibility and efficient access to application semantics.

Each of these categories offers a different tradeoff between power, runtime cost, ease of use, and ease of modification. For example, declarative notations are the easiest to use (especially for nonprogramming user interface designers) but have the least power, since they can only represent a predetermined range of possibilities. Typically, several notational techniques are used in a system, with different aspects of the user interface controlled by different techniques. For example, the position and size of a screen button might be specified graphically, while its highlighting behavior is specified implicitly by the code of a toolkit routine.

Application control flow indicates where input processing occurs in the application's flow of control:

- **Single input point:** The system contains an *event loop* that is the sole point at which user input is accepted; when an input event is received, it is processed; then control returns to the event loop to await the next input. Note that the event loop may be in either application or shared code.
- **Multiple input point:** Input is accepted at multiple points in the application's flow of control. (Usually, each such point can handle only a subset of the possible inputs, leading to modal interface behavior.)

This classification is a variation of the standard distinction between "internal control" (application calls user interface) and "external control" (user interface calls application) [Hayes 85, Tanner 83]. The standard terminology is unsatisfactory because the properties usually associated with external control actually apply to any system using an event loop, regardless of the direction of subroutine calls.

Number of control threads indicates how many logical threads of control exist in the application and user interface:

- **Single thread of control.**
- **One user interface thread and one application thread.**
- **Multiple user interface threads and one application thread.**
- **One user interface thread and multiple application threads.**
- **Multiple user interface threads and multiple application threads.**

Multiple threads are useful for dealing with external events or logically independent concurrent dialogues (e.g., multiple input devices). The one-plus-one-thread choice is particularly simple and helpful for decoupling application processing (including external event handling) from user interface logic.

Control thread mechanism describes the method, if any, used to support multiple logical threads of control. Often, full-fledged processes are too difficult to implement or impose too much overhead, so many partial implementations are used. This dimension classifies the possibilities as follows:

- **None:** Only a single logical control thread is used.
- **Standard processes:** Independently scheduled entities with interprocess protection (typically, separate address spaces). These provide security against other processes, but interprocess communication is relatively expensive. For a user interface system, security may or may not be a concern, while communication costs are almost always a major concern. In network environments, standard processes are usually the only kind that can be executed on different machines.
- **Lightweight processes:** Independently scheduled entities within a shared address space. These are only suitable for mutually trusting processes due to lack of security; but often that is not a problem for user interface systems. The benefit is substantially reduced cost of communication, especially for use of shared variables. Few operating systems provide lightweight processes, and building one's own lightweight process mechanism can be difficult.
- **Non-preemptive processes:** Processes without preemptive scheduling (must explicitly yield control), usually in a shared address space. These are relatively simple to implement. Guaranteeing short response time is difficult and impacts the entire system: long computations must be broken up explicitly.
- **Event handlers:** Pseudo-processes which are invoked via a series of subroutine calls; each such call must return before another event handler process can be executed. Hence control flow is restricted; in particular, waiting for another process cannot occur inside a subroutine called by an event handler. Again, response time constraints require system-wide attention. The main advantage of this method is that it requires virtually no support mechanism.
- **Interrupt service routines:** Hardware-level event handling; a series of interrupt service routine executions form a control thread, but one with restricted control flow and communication abilities. The control flow restrictions are comparable to event handlers; but unlike event handlers, preemptive scheduling is available.

Event handlers are easily implemented within a user interface system; non-preemptive processes are harder but can still be implemented without operating system support. The other mechanisms usually must be provided by the operating system. Some form of preemptive scheduling is often desirable to reduce timing dependencies between threads.

Basis of communication classifies systems according to whether communication between modules depends upon shared state, upon events, or both. An *event* is a transfer of information occurring at a discrete time, for example, via a procedure call or message. Communication through shared state variables is significantly different, because the recipient always has access to the current values and need not use information in the same order in which it is sent. The design space recognizes four categories:

- **Events:** There is no shared state; all communication relies on events.
- **Pure state:** Communication is strictly via shared state; the recipient must repeatedly inspect the state variables to detect changes.
- **State with hints:** Communication is via shared state, but the recipient is actively informed of changes via an event mechanism; hence polling of the state is not required. However, the recipient could ignore the events and reconstruct all necessary information from the shared state; so the events are efficiency hints rather than essential information.
- **State plus events:** Both shared state and events are used; the events are crucial because they provide information not available from state monitoring.

State-based mechanisms are popular for dealing with incrementally updated displays. The hybrid state/event categories provide possibilities for performance optimization in return for their extra complexity. State-based communication requires access to shared storage, which may be impossible or unreasonably expensive in some system architectures.

It is possible for different bases of communication to be used at the application and device interfaces, but this is rare. It is fairly common to have different bases of communication for input and output; hence the design space provides separate dimensions for input and output communication basis.

3. Design Rules for User Interface Systems

This section presents some design rules that relate the functional and structural dimensions of the design space. Again, we will consider only a few sample rules to illustrate the flavor of the approach. For a more thorough presentation of the rules, see Appendix B.

Both here and in Appendix B, we present the rules in an informal fashion. In this form, the rules are directly useful as guidelines or rules of thumb for a human designer. Section 4 describes how the rules can be made more formal and suitable for use in automated design.

- Stronger requirements for user interface adaptability across devices favor higher levels of application interface abstraction, so as to decouple the application from user interface details that may change across devices. If the requirement is for global behavior or application semantics changes, then parameterized abstract devices are also favored. Such changes generally have to be implemented in shared user interface code or application code, rather than in the device driver; so information about the device at hand cannot be hidden from the higher levels, as the other classes of abstract device try to do. However, a requirement for local behavior changes can favor abstract devices with variable operations, since this method can allow all of the required adaptation to be hidden within the device driver.
- High user customizability requirements favor external notations for user interface behavior. Implicit and internal notations are usually more difficult to access and more closely coupled to application logic than are external notations.
- A high requirement for application portability across user interface styles favors the higher levels of application interface abstraction. Less obviously, it favors event-based or pure state-based communication over the hybrid forms (state with hints or state plus events). A hybrid communication protocol is normally tuned to particular communication patterns, which may change when user interface style changes.
- If the maximum command execution time is short, a single thread of control is practical and is favored as the simplest solution. With longer commands, multiple threads are favored to permit user input processing to continue; this is necessary to support command cancellation, for example.
- If external events must be handled, it is often worthwhile to provide separate control thread(s) for this purpose. Separate threads serve to decouple event handling logic from user interface logic. When external event handling requires preemption of user commands, a preemptive control thread mechanism (standard processes, lightweight processes, or interrupt service routines) is strongly favored. Without such a mechanism, very severe constraints must be placed on all user interface and application processing in order to guarantee adequate response time.
- The most commonly useful control thread mechanisms are standard processes, lightweight processes, and event handlers; the others are appropriate only in special cases. For most user interface work, lightweight processes are very appropriate if available. Standard processes should be used when protection considerations warrant, and in network environments where it may be useful to

put the processes on separate machines. If these conditions do not apply, event handlers are the best choice when their response time limitations are acceptable; otherwise it is probably best to invest in building a lightweight process mechanism.

The preceding rules all relate functional to structural dimensions. Following is an example of the rules interconnecting structural dimensions.

- The choice of application interface abstraction level influences the choice of notation for user interface behavior. In monolithic programs and abstract-device application interfaces, implicit representation is usually sufficient. In toolkit systems, implicit and internal declarative notations are found (parameters to toolkit routines being of the latter class). Interaction managers of all types use external and/or internal declarative notations. Extensible interaction managers rely heavily on procedural notations, particularly internal procedural notation, since customization is often done by supplying procedures.

The reader may well have found these rules to be fairly obvious and a bit boring. This is an indication of the conceptual power of the design space: many useful rules are immediate consequences of the properties of the chosen dimensions. Though straightforward, these rules are sufficiently powerful to be a useful aid to design.

4. Automating the Design Rules

The design rules are presented in this report in an informal fashion suitable for use as guidelines by human software designers. It is also possible to express the rules in a more detailed, rigorous formulation. In such a form the rules could be used as the basis for an automatic design aid. The author has experimented with such automated rules, with promising results.

The rules were expressed in the form of numerical weights associated with particular combinations of values along different dimensions. For example, the combination of no external events and single thread of control received a positive weight, indicating that a single thread of control may be a good choice given that requirement; while the combination of preemptive external events and single thread of control received a negative weight. Given a set of functional requirements and a proposed structural design, the weights of the applicable rules can be combined to give a score for that design. A straightforward search algorithm was used to find the highest-scoring design for a given set of requirements.

These automated rules were tested by comparing their recommendations to the actual design choices of expert human software designers, as expressed in a set of test cases. A moderate to substantial degree of agreement was observed. This preliminary result suggests that this approach has considerable potential for creating practical design aids. More immediately, it gives some confidence that the design space described here captures useful knowledge about user interface software design.

Additional information about this experiment is given in the companion report [Lane 90b]. For full details, see [Lane 90a].

5. Summary

This design space is directly usable as a notation for describing and comparing user interface system architectures. It should be useful for both the design and understanding of systems. The design rules provide a good starting point for the process of user interface structural design. As presented, the rules have been simplified too much to be capable of making subtle tradeoffs, but they can still help a designer to identify the better alternatives and to reject inappropriate structures. By reducing the mental effort needed to make the straightforward choices, these rules should free the designer to concentrate on the hard choices.

An automated form of the design rules has shown a substantial degree of agreement with the choices of human designers. One important implication of this result is that the design space provides considerable conceptual leverage: the space is "right" in the sense that using it makes choosing an appropriate design easier.

The design space and rules described here were based on an extensive survey of existing user interface systems [Lane 90a]. The space was formed by searching for classifications that brought systems with similar properties together. The rules were then prepared on the basis of observed correlations. This process can be compared to development of biological taxonomies through natural history: the biologist also surveys and classifies existing forms, then looks for explanatory theories.

At present there is no theoretical basis on which to argue that this design space is better or worse than a different set of dimensions that might be constructed to describe the same systems. The design space can be defended only on the grounds of practical utility: it seems to capture some useful design ideas and correlations. Further experience and research will no doubt improve this space, and someday a more theoretical, rigorous basis for creating design spaces may emerge.

Future work includes refining the design space and rules to cover lower-level choices, thus providing more detailed design advice. A full-scale attempt to automate the rules might produce a practical design aid. In the long term, we hope that this work can be generalized to yield principles of software architecture that hold beyond the domain of user interfaces.

Appendix A: The Design Space

This appendix provides a full description of the design space used in the experiment with automated rules. This space is probably somewhat different from what one would use in hand design work.

The design space contains twenty-five functional dimensions. Three to five alternatives are recognized in each of these dimensions. There are nineteen structural dimensions, each offering two to seven alternatives.

A.1. Functional Design Dimensions

We turn first to the functional design dimensions, which identify the requirements for a user interface system that most affect its structure. These dimensions fall into three groups:

- **External requirements:** Includes requirements of the particular applications, users, and I/O devices to be supported, as well as constraints imposed by the surrounding computer system.
- **Basic interactive behavior:** Includes the key decisions about user interface behavior that fundamentally influence internal structure.
- **Practical considerations:** Cover development cost considerations; primarily, the required degree of adaptability of the system.

A.1.1. External Requirements

A.1.1.1. Application Characteristics

The characteristics of the problem domain determine the features needed to provide an adequate user interface for a particular application or set of applications. A general-purpose user interface system may support more than one of the alternatives listed for any of these dimensions.

Primary output capability. What will be the system's main means of communicating information to its user? We classify the alternatives according to the type of data presented:

- **Text:** Displayed character strings.
- **Geometric graphics:** Images describable by geometric elements (lines, circles, etc). For example, engineering drawings.
- **General images:** Images not readily described by geometric elements, such as scanned photographs or bitmap artwork.
- **Voice:** Audible speech.
- **Audio:** Non-speech audible output, such as music or tonal signals.

Primary input capability. What is the system's main method of receiving information from its user?

- **Discrete selection:** Selection of one of a small set of alternatives; for example, selection from a menu, or a "yes/no" response.
- **Continuous selection:** Selection of a point in some continuum; for instance "pointing" to a point on a display surface, or manipulating a slider or control dial.
- **Text:** Textual data, usually typed on a keyboard; this is distinguished from discrete selection by a wider set of permissible inputs. (For instance, if the user is required to press `Y` or `N` to answer "yes" or "no," that is discrete selection via a keyboard; but entry of prose into a word processor, or names and addresses into a mailing list database, is textual input.)
- **Voice, discrete words:** Words are recognized individually, without use of grammar or context information.
- **Voice, connected speech:** Full-fledged speech recognition, using semantic context information to distinguish ambiguous words.

Command execution time. How long may the application program take to process a command, compared with the reaction time of a human user?

- **Short maximum time:** All commands can be executed in a short time, say a few tenths of a second.
- **Intermediate maximum, short average:** Most commands are executed in a short time, but some may take a bit longer, up to a couple of seconds.
- **Long maximum time:** Some or all commands may take a long time to execute so that the user will have a strong perception of waiting.

External event handling. Does the application program need to respond to external events, that is, events not originating in the user interface? If so, on what time scale?

- **No external events:** The application is uninfluenced by external events, or checks for them only as part of executing specific user commands. For example, a mail program might check for new mail, but only when an explicit command to do so is given. In this case, no support for external events is needed in the user interface.
- **Process events while waiting for input:** The application must handle external events, but response time requirements are not so stringent that it must interrupt processing of user commands. It is sufficient for the user interface to allow response to external events while waiting for input.
- **External events preempt user commands:** External event servicing has sufficiently high priority that user command execution must be interrupted when an external event occurs.

Error prevention importance. How important is prevention of user error, relative to other goals (such as speed of operation)?

- **High:** Error prevention is critical to the task (e.g., automated banking).

- **Medium:** Error prevention is of intermediate importance.
- **Low:** Error prevention is a minor issue.

A.1.1.2. User Needs

What features are needed for the intended user community? The dimensions affecting system structure are:

User help needs. How much user assistance is provided?

- **High:** Extensive assistance for novices is provided.
- **Medium:** Some guidance for novices is provided.
- **Low:** User interface is oriented towards expert users.

User experience variability. How much variability in experience is catered for?

- **High:** Different user interfaces are provided for novice and expert users.
- **Medium:** Minor changes in behavior are available for expert users.
- **Low:** No adaptation to different experience levels is provided.

User customizability. How much can a user modify the system’s behavior? (We have in mind end users, not application developers.)

- **High:** User can add new commands and redefine commands (e.g., via a macro language), as well as modify user interface details.
- **Medium:** User can modify details of the user interface that do not affect semantics; for instance, change menu entry wording, default window sizes, colors, etc.
- **Low:** Little or no user customizability.

A.1.1.3. I/O Devices

What types of I/O devices will be used for communication with the user? The crucial aspects for system structure are:

Device class breadth. What range of I/O devices is supported by the user interface software? (We are interested here in the range of devices that are considered equivalent at some level of the software; for example, if two different displays are supported, they are probably equivalent at some level, but a display and a speaker would probably not be considered equivalent.)

- **Single device type:** Only a specific hardware type is permitted.
- **Semantically equivalent devices:** Devices with a fixed set of features are permitted; for example, 24x80 character terminals with cursor positioning and underlining capability. Any additional features possessed by a particular device are ignored. The means of invoking the required features may vary between devices.

- **Generic device definition:** A wide range of devices is permitted; for example, alphanumeric terminals of varying size with optional color and highlighting capabilities.

User interface adaptability across devices. How much change in user interface behavior may be required when changing to a different set of I/O devices?

- **None:** All aspects of behavior are the same across all supported devices.
- **Local behavior changes:** Only changes in small details of behavior across devices; for example, the appearance of menus.
- **Global behavior changes:** Major changes in surface user interface behavior; for example, a change in basic interface class (see below).
- **Application semantics changes:** Changes in underlying semantics of commands (e.g., continuous display of state versus display on command).

I/O device bandwidth. What data rate is needed to support the user interface I/O devices? (For devices with persistent state such as displays, use the burst rate needed for updates.)

- **High:** Kilobytes per second (e.g., high-resolution bitmap displays).
- **Medium:** Hundreds of bytes per second (e.g., alphanumeric terminals).
- **Low:** Tens of bytes per second (e.g., teletypes or small LED displays).

A.1.1.4. Computer System Environment

The surrounding computer system affects a user interface in several ways. The key issues are:

Strength of user interface conventions. How strong are the user interface conventions of the computer system?

- **High:** Extensive, well-defined standards which are generally followed (e.g., the Macintosh user interface guidelines [Apple 85]).
- **Medium:** Conventions exist but are incomplete and/or often violated (e.g., Unix conventions for command line syntax).
- **Low:** Little or no recognized common user interface behavior. (This is the situation for many stand-alone systems, such as automated store directories.)

Inter-application communication requirements. What kind of inter-application communication is supported *by the user interface*? ("Back door" communication such as data file exchange is not counted.)

- **None:** No communication at the user interface level.
- **Data exchange:** Via cut-and-paste or standardized I/O formats.
- **Program invokes program:** One program drives another, issuing commands

and interpreting responses. (Examples include Unix shell scripts and various macro languages.)

Inter-application protection requirements. To what extent does shared user interface software provide protection boundaries between different applications?

- **High:** User interface deals with multiple applications and must prevent undesirable interactions.
- **Medium:** User interface deals with multiple applications, but only weak protection is needed (e.g., applications are expected to cooperate).
- **Low:** No protection is needed (typically because user interface deals with only one application at a time).

Computer system organization. What is the overall organization of the computer system?

- **Uniprocessing:** A single application executes at a time.
- **Multiprocessing:** Multiple applications execute concurrently.
- **Distributed processing:** Network environment, with multiple CPUs and non-negligible communication costs.

Existing mechanisms for multiple threads of control. Does the operating system provide any mechanism(s) for multiple control threads?

- **Standard processes:** Independently scheduled entities with interprocess protection (typically, separate address spaces).
- **Lightweight processes:** Independently scheduled entities with no interprocess protection (shared address space).
- **Non-preemptive processes:** Processes without preemptive scheduling (must explicitly yield control); usually no interprocess protection.
- **Interrupt service routines:** Hardware-level event handling (a series of interrupt service routine executions can be viewed as a control thread).
- **None:** No system support for multiple control threads.

Processing power available for user interface. Is adequate processing power available for the user interface, or is it necessary to "cut corners" in the system design to achieve adequate response time?

- **High:** Plenty of processing power is available.
- **Medium:** Some care is needed to achieve adequate performance.
- **Low:** Must minimize resources used by user interface.

Designers usually make a rough judgment about available power at a fairly early stage in the design process, and this judgment colors many subsequent decisions. We include this dimension in the design space to make this judgment explicit.

A.1.2. Basic Interactive Behavior

This group of dimensions includes the key decisions about user interface behavior that fundamentally influence internal structure. Fortunately these are few; otherwise a single structure could not support a range of interaction styles.

Basic interface class. This dimension identifies the basic kind of interaction supported by the user interface system. (A general-purpose system might support more than one of these classes.) The design space uses a classification proposed by Shneiderman [Shneiderman 86]:

- **Menu selection:** Based on repeated selection from groups of alternatives; at each step, the alternatives are (or can be) displayed.
- **Form filling:** Based on entry (usually text entry) of values for a given set of variables.
- **Command language:** Based on an artificial, symbolic language; often allows extension through programming-language-like procedure definitions.
- **Natural language:** Based on (a subset of) a human language such as English.
- **Direct manipulation:** Based on direct graphical representation and incremental manipulation of the program's data.

It turns out that menu selection and form filling can be supported by similar system structures, but each of the other classes has unique requirements.

Degree of user control over dialog sequence. How much control does the user have over the sequence of interactions with the system?

- **High:** User controls dialog sequence (e.g., "modeless" dialog).
- **Medium:** User has some control over dialog.
- **Low:** Machine controls dialog sequence.

A.1.3. Practical Considerations

The remaining functional dimensions specify the required degree of adaptability of the system. In most cases a less adaptable system is cheaper to build. Yet a more adaptable system may repay its higher cost by supporting a wider class of applications. Another important consideration is that a system's adaptability affects its maintainability, and hence its lifespan.

It is useful to consider adaptability separately for application code and user interface code. The distinction disappears in single-purpose user interfaces, but is crucial for user interface systems that support multiple applications. We use the term *portability* for application code and *adaptability* for user interface code. This terminology is intended to connote the idea that we usually desire application code not to change when moving from one environment to another, while user interface support systems may well be modified to better adapt them to new environments. (Of course there are exceptions to this general rule.) Portability implies that the application is unaware of a change in environment, or at least can handle the change without being rewritten.

Application portability across I/O devices. What degree of portability across I/O devices is required for applications that will use the user interface software?

- **High:** Applications should be portable across devices of radically different types; for example, display versus speech output.
- **Medium:** Applications should be portable across devices of the same general class, but differing in detail; for example, bitmap displays of differing color capabilities.
- **Low:** Device independence is not a concern, or application changes are acceptable to support new devices.

Application portability across interaction styles. What degree of portability across user interface styles is required for applications that will use the user interface software?

- **High:** Applications should be portable across significantly different styles (e.g., command language versus menu-driven).
- **Medium:** Applications should be independent of minor stylistic variations (e.g., menu appearance).
- **Low:** User interface variability is not a concern, or application changes are acceptable when modifying the user interface.

Application portability across operating systems. What degree of portability across underlying computer systems is required for applications that will use the user interface software? (Primarily we are interested in operating system differences, though hardware differences may also be of interest.)

- **High:** Applications should be portable across significantly different machines and operating systems.
- **Medium:** Applications should be portable across related operating systems (e.g., portable to different versions of Unix).
- **Low:** System independence is not a concern.

User interface system adaptability across applications. How adaptable to different applications should the user interface software be?

- **High:** Useful across a wide range of applications.
- **Medium:** Useful for a group of closely related applications with similar interface needs.
- **Low:** Supports only a single application.

User interface system adaptability across interaction styles. How adaptable to different interaction styles should the user interface software be?

- **High:** Adaptable to a wide range of interface styles.
- **Medium:** Limited adaptability.
- **Low:** Imposes a specific interface style.

A user interface system may well be built to impose some stylistic decisions on applications; it is by no means the case that more flexibility is always better.

User interface system adaptability across operating systems. How adaptable to different computer systems should the user interface software be?

- **High:** Portable across significantly different machines and operating systems.
- **Medium:** Portable across related operating systems.
- **Low:** System independence is not a concern.

A.2. Structural Design Dimensions

We now turn to the structural dimensions, which represent the major decisions determining the overall structure of a user interface system. These dimensions fall into three major groups:

- **Division of functions and knowledge between modules:** How system functions are divided into modules, the interfaces between modules, and the information contained within each module.
- **Representation issues:** The data representations used within the system. We must consider both actual data, in the sense of values passing through the user interface, and *meta-data* that specifies the appearance and behavior of the user interface. Meta-data may exist explicitly in the system (for example, as a data structure describing the layout of a dialog window), or only implicitly.
- **Control flow, communication, and synchronization issues:** The dynamic behavior of the user interface code.

The structural design space presented here is a simplification of the complete design space discussed in [Lane 90a]. The simplification arises primarily from merging together decisions that proved to be closely correlated in practice. We will mention some of the omitted dimensions under the headings of the key dimensions with which they are associated.

A.2.1. Division of Functions and Knowledge

Under this heading, we consider how system functions are divided into modules, the interfaces between modules, and the information contained within each module.

The divisions of greatest interest are the divisions between application-specific code and shared user interface code on the one hand, and between device-specific code and shared user interface code on the other. We refer to these divisions as the *application interface* and *device interface*, respectively. (See Figure 2-1.)

Application interface abstraction level. The design space identifies six general classes of application interface. These classes can be most easily distinguished by their level of abstraction:

- **Monolithic program:** There is no separation between application-specific and shared code, hence no application interface (and no device interface, either).
- **Abstract device:** The shared code is simply a device driver, presenting an abstract device for manipulation by the application. The operations provided have specific physical interpretations (e.g., "draw line," but not "present menu"). Most aspects of interactive behavior are under the control of the application, although some local interactions may be handled by the shared code (e.g., character echoing and backspace handling in a keyboard/display driver). In this category, the application interface and device interface are the same.
- **Toolkit:** The shared code provides a library of interaction techniques (e.g., menu or scroll bar handlers). The application is responsible for selecting ap-

propriate toolkit elements and composing them into a complete interface; hence the shared code can control only local aspects of user interface style, with global behavior remaining under application control. The interaction between application and shared code is in terms of specific interactive techniques (e.g., "obtain menu selection"). The application can bypass the toolkit, reaching down to an underlying abstract device level, if it requires an interaction technique not provided by the toolkit. In particular, conversions between specialized application data types and their device-oriented representations are done by the application, accessing the underlying abstract device directly.

- **Interaction manager with fixed data types:** The shared code controls both local and global interaction sequences and stylistic decisions. Its interaction with the application is expressed in terms of abstract information transfers, such as "get command" or "present result" (notice that no particular external representation is implied). These abstract transfers use a fixed set of standard data types (e.g., integers, strings); the application must express its input and output in terms of the standard data types. Hence some aspects of the conversion between application internal data formats and user-visible representations remain in the application code.
- **Interaction manager with extensible data types:** As above, but the set of data types used for abstract communication can be extended. The application does so by specifying (in some notation) the input and output conversions required for the new data types. If properly used, this approach allows knowledge of the external representation to be separated from the main body of the application.
- **Extensible interaction manager:** Communication between the application and shared code is again in terms of abstract information transfers. The interaction manager provides extensive opportunities for application-specific customization. This is accomplished by supplying code that augments or overrides selected internal operations of the interaction manager. (Most existing systems of this class are coded in an object-oriented language, and the language's inheritance mechanism is used to control customization.) Usually a significant body of application-specific code customizes the interaction manager; this code is much more tightly coupled to the internal details of the interaction manager than is the case with clients of nonextensible interaction managers.

This classification turns out to be sufficient to predict most aspects of the application interface, including the division of user interface functions, the type and extent of application knowledge made available to the shared user interface code, and the kinds of data types used in communication. For instance, we have already suggested the division of local versus global control of interactive behavior that is typically found in each category.

Variability in device-dependent interface. The interface between device-dependent and device-independent code can be regarded as defining an **abstract device** for the device-independent code to manipulate. This dimension classifies abstract devices according to the degree of variability perceived by the device-independent code.

- **Ideal device:** The provided operations and their results are well specified in terms of an "ideal" device; the real device is expected to approximate the ideal behavior fairly closely.

- **Parameterized device:** A class of devices is covered, differing in specified parameters such as screen size, number of colors, number of mouse buttons, etc. The device-independent code can inquire about the parameter values for the particular device at hand, and adapt its behavior as necessary. Operations and their results are well specified, but depend on parameter values.
- **Device with variable operations:** A well-defined set of device operations exists, but the device-dependent code has considerable leeway in choosing how to implement the operations; device-independent code is discouraged from being closely concerned with the exact external behavior. Results of operations are thus not well specified.
- **Ad-hoc device:** In many real systems, the abstract device definition has developed in an ad-hoc fashion, and so it is not tightly specified; behavior varies from device to device. Applications therefore must confine themselves to a rather small set of device semantics if they wish to achieve portability, even though any particular implementation of the abstract device may provide many additional features.

The reader may wonder why there is no dimension that classifies abstract devices according to their basic functionality. Such a dimension might use categories like "bitmap display," "vector display," "alphanumeric display," "keyboard," "two-dimensional locator," etc. But there are a large number of such categories, with no obvious pattern. Moreover, much of the useful information has already been captured in other dimensions (device bandwidth, primary input and output capability). The simplified design space therefore provides no such dimension.

A.2.2. Representation of Information

Here we consider the representations used for user interface data. Since we are studying overall system structure, we are more interested in representations that are shared among modules than in those that are hidden within a single module.

Notation for user interface definition. This dimension classifies the techniques used for defining user interface appearance and behavior.

- **Implicit in shared user interface code:** Information buried within shared code. For example, the visual appearance of a menu might be implicit in the menu routines supplied by a toolkit.
- **Implicit in application code:** Information buried in the application and not readily available to shared user interface code.
- **External declarative notation:** A nonprocedural specification separate from the body of the application program, for example, a grammar or tabular specification. **Graphical specification** is an important special case, particularly useful for specification of visual appearance.
- **External procedural notation:** A procedural specification separate from the body of the application program; often cast in a specialized programming language.

- **Internal declarative notation:** A nonprocedural specification within the application program. This differs from an implicit representation in that it is available for use by the shared user interface code. Parameters supplied to user interface library routines often amount to an internal declarative notation. An example is a list of menu entries provided to a toolkit menu routine.
- **Internal procedural notation:** A procedural specification within the application program. This differs from an implicit representation in that it is available for use by the shared user interface code. A typical example is a status-inquiry or data transformation function that is provided for the user interface code to call.

Representation of semantic information. This dimension classifies the techniques used for defining application-specific semantic (as opposed to external appearance) information that is needed by the user interface. An example of such information is a range restriction on an input value.

- **Implicit:** Buried in the application, and not readily available to shared user interface code. For example, a range check carried out as part of command execution.
- **Declarative:** Expressed in a nonprocedural notation; for example, a form-filling package might allow range limits to be given in a table entry describing a numeric input field.
- **Procedural:** A procedural specification within the application program. This differs from an implicit representation in that it is available for use by the shared user interface code. For example, a validity checking subroutine might be provided for each input value.

The limited range of possibilities allowed by a declarative notation is more of a drawback here than it is for user interface definition. (Semantic information is inherently more variable across applications than surface user interface choices; were this not so, shared user interface behavior would be of no interest.) Procedural representations are therefore commonly used where shared code must have access to semantic information, while implicit representations are used where this can be avoided.

A.2.3. Control Flow, Communication, and Synchronization

Here we consider the dynamic behavior of the user interface code. As with the previous group of dimensions, we are mainly interested in inter-module communication.

Application control flow. Where does input processing occur in the application's flow of control?

- **Single input point:** The system contains an *event loop* that is the sole point at which user input is accepted; when an input event is received, it is processed; then control returns to the event loop to await the next input. Note that the event loop may be in either application or shared code.
- **Multiple input point:** Input is accepted at multiple points in the application's flow of control. (Usually, each such point can handle only a subset of the possible inputs, leading to modal interface behavior.)

This classification is a variation of the standard distinction between "internal control" (application calls user interface) and "external control" (user interface calls application) [Hayes 85, Tanner 83]. The standard terminology is unsatisfactory because the properties usually associated with external control actually apply to any system using an event loop, regardless of the direction of subroutine calls.

Treatment of asynchronous input. What happens to user input actions that occur while the application is busy?

- **Ignored:** Asynchronous input is ignored.
- **Queue before all processing:** Input events are queued, but no processing is done (and hence no feedback occurs) until the application is ready for input.
- **Partial processing, simple queue:** Some fast processing is done to provide feedback; then events are queued for the application in a first-in-first-out queue.
- **Partial processing, complex queue:** As above, but the queue may not be strictly FIFO; for instance, "abort" commands may be delivered first, or may flush the queue.

Note that the first two of these alternatives correspond to no fast input processing, while the second two describe systems which have some type of fast input processing.

Fast input processing. Is user input processed before the application is ready to receive it? If so, how flexible is this processing?

- **No fast processing:** Everything is synchronous with the application.
- **Fixed behavior:** Some processing and feedback is done asynchronously; the nature of the asynchronous processing is not alterable by the application. (Example: input echoing and editing in older time-sharing systems.)
- **Parameterized behavior:** Application-specific code can set limited parameters for the behavior of the asynchronous processing. For example, in some window systems, different cursor shapes can be established for different parts of an application's window. Shape changes are then handled automatically by the cursor tracking code.
- **Application-dependent behavior:** Application-specific code can be executed during fast processing. For example, an application-specific routine might be used to draw rubber-band feedback images during dragging.

The more flexible alternatives in this dimension carry increasing risk of synchronization problems. (A simple example is that typed-ahead characters may be echoed twice or not at all when switching between asynchronous echoing and application-driven echoing.) Communication costs can also be a problem for the last alternative.

Number of control threads. How many control threads exist in the application and user interface?

- **Single thread of control.**

- **One user interface thread and one application thread.**
- **Multiple user interface threads and one application thread.**
- **One user interface thread and multiple application threads.**
- **Multiple user interface threads and multiple application threads.**

Multiple threads are useful for dealing with external events or logically independent concurrent dialogues (e.g., multiple input devices). The one-plus-one-thread choice is particularly simple and helpful for decoupling application processing (including external event handling) from user interface logic.

Control thread mechanism. What mechanism, if any, is used to support multiple control threads?

- **None:** Only a single logical control thread is used.
- **Standard processes:** Independently scheduled entities with interprocess protection (typically, separate address spaces).
- **Lightweight processes:** Independently scheduled entities within a shared address space.
- **Non-preemptive processes:** Processes without preemptive scheduling (must explicitly yield control), usually in a shared address space.
- **Event handlers:** Pseudo-processes which are invoked via a series of subroutine calls; each such call must return before another event handler process can be executed.
- **Interrupt service routines:** Hardware-level event handling; a series of interrupt service routine executions form a control thread, but one with restricted control flow and communication abilities. Unlike simple event handlers, preemptive scheduling is available.

Application communication grain size. How frequently does communication occur between application and shared user interface code?

- **Fine grain:** Roughly once per user input event; the application is closely coupled to user actions, and typically participates in feedback generation.
- **Coarse grain:** Roughly once per complete command; the application is decoupled from user actions and feedback generation.

Either of these approaches may be preferable, depending on the desired extent of application involvement in user interface details.

Device communication grain size. How frequently does communication occur between device-independent and device-dependent code?

- **Fine grain:** Roughly once per physical input event; the device-independent code is involved in generating short-term feedback displays.

- **Coarse grain:** Roughly once per logical interaction; the device-independent code is not involved in short-term feedback generation.

Basis of communication. Does communication between modules depend on shared state or on events, or both? (An *event* is a transfer of information occuring at a discrete time, for example, via a procedure call or message.)

- **Events:** There is no shared state; all communication relies on events.
- **Pure state:** Communication is strictly via shared state; the recipient must repeatedly inspect the state variables to detect changes.
- **State with hints:** Communication is via shared state, but the recipient is actively informed of changes via an event mechanism; hence polling of the state is not required. However, the recipient could ignore the events and reconstruct all necessary information from the shared state; so the events are efficiency hints rather than essential information.
- **State plus events:** Both shared state and events are used; the events are crucial because they provide information not available from state monitoring.

It is possible for different bases of communication to be used at the application and device interfaces, but this is rare. It is fairly common to have different bases of communication for input and output; hence the design space provides separate dimensions for input and output communication basis.

Event mechanisms. Unless pure-state communication is used, a mechanism must be provided to pass events between modules. We classify event mechanisms thus:

- **None:** No events are used (pure state communication).
- **Direct procedure call:** Standard procedure-call mechanism. (We include "remote procedure call" mechanisms, so long as the recipient code is directly named.)
- **Indirect procedure call:** Procedure call in which the recipient code is not completely specified by the calling code, but is dynamically determined; procedure pointers and object-oriented method calls are typical examples.
- **Asynchronous message:** The event is passed from one control thread to another, with the sender not waiting for receipt.
- **Synchronous message:** The event is passed from one control thread to another, with the sender blocked until the receiver accepts the message (and computes a reply, usually). This differs from a remote procedure call in that the receiver is a separate control thread that exists before and after the rendezvous.

The procedure call mechanisms are used for communication within a control thread, the message mechanisms for communication across threads. Indirect procedure calls provide extra separation at slightly higher cost. Synchronous message mechanisms are somewhat cheaper to implement than asynchronous ones (for instance, message buffering can be avoided), but they may create synchronization problems by increasing timing dependencies between control threads.

It is common to have different event mechanisms for input and output, and also to have different mechanisms at the application and device interfaces. Hence the design space provides four event-mechanism dimensions, one each for application input, application output, device input, and device output.

Application separation mechanism. How strongly are the application and shared user interface code separated?

- **Programming convention:** No mechanism exists to enforce separation.
- **Visibility rules:** A programming language mechanism such as separate name spaces. Protection strength depends on whether the language is secure against errors (such as dangling pointers).
- **Hardware separation:** A hardware mechanism, typically separate address spaces. The shared user interface code is reliably protected against programming errors in the application (and vice versa).
- **Network link:** In addition to providing hardware separation, the communication protocol allows for cross-machine communication; data representation differences between application and user interface code are supported. An example is the support for varying byte order in the X Window protocol.

These choices provide a tradeoff of security against cost of communication. The availability of suitable mechanisms is also a consideration; many small machines do not provide hardware protection mechanisms.

Device separation mechanism. How strongly are the device-dependent and device-independent layers separated?

The classification is the same as for the previous dimension.

The data volume and frequency of communication are usually higher here than at the application interface, so the cost of communication is a greater concern. Thus a different choice is often appropriate.

Appendix B: The Design Rules

This appendix presents some simple "rules of thumb" that help a designer of user interface software to select a system architecture. These rules are not meant to replace good design judgment, but rather to codify and speed up the routine parts of system design. The rules let the designer make quick decisions about aspects of system structure for which there is a clearly superior alternative, and they focus attention on the most likely choices in cases where more subtle judgment is necessary.

We discuss the design dimensions in the order in which a designer might consider them while creating a design. For each dimension, we present a listing of the considerations that may favor or disfavor each alternative, and some summary rules-of-thumb for selecting one alternative. Again we emphasize that these rules must be augmented by the designer's judgment: typically, the designer must resolve conflicting suggestions by judging the relative importance of different functional requirements.

Space limitations prohibit any attempt to provide justifications of these observations and rules. Supporting arguments can be found in [Lane 90a].

B.1. Basic Division of Functions

The designer's first order of business should be to define the overall division of a system into device-specific, shared user interface, and application-specific parts. We view this as a problem of specifying two interfaces: the *application interface* between application-specific and shared code, and the *device interface* between device-specific and shared code.

B.1.1. Application Interface

Application interface abstraction level. The design space identifies six general classes of application interface. In order of increasing level of abstraction in communications, they are:

- **Monolithic program:** This is an appropriate solution in small, specialized systems where the application needs considerable control over UI details and/or little processing power is available. (Video games are a typical example.) This approach should not be chosen if there are any strong flexibility requirements (user customizability, I/O device variability, or UI style flexibility). The approach handles direct manipulation interfaces well, but application development effort will be high.
- **Abstract device:** This approach is recommended when application portability is wanted across a limited set of devices, but most control of the user interface is to remain in the hands of the application program. Thus it is not a good choice when application portability across UI styles is a strong requirement. It is best not to attempt to support a very wide range of I/O devices with this approach; the result will be either excess development effort for applications (too many cases to handle) or loss of control over UI details (if the driver hides too

many details). The characteristics of this approach are heavily influenced by the handling of abstract device variability, which is discussed in Section B.1.2.

- **Toolkit:** Toolkits provide a significant savings of application development effort, and yet retain UI system flexibility since the application remains "in charge" and can bypass the toolkit when necessary. By the same token, the application remains coupled to the user interface. Therefore, this approach is recommended when a moderate degree of flexibility is wanted. This approach is the minimum level of abstraction to use when a standardized UI style is to be implemented, because standard components (e.g., menus) can be handled by toolkit routines rather than reimplemented by each application.
- **Interaction manager (IM):** An IM is a good choice when application portability (across devices or styles) is a strong requirement, because it provides a strong separation between UI behavior and the application program. A high degree of user customizability can also be supported. However, supporting direct manipulation interfaces is difficult because the application cannot supply semantic feedback. An IM is useful for enforcing standardized UI behavior, since it gives the application program the least control over UI details of any alternative. An IM is especially appropriate in network environments, because the IM can be physically separated from the application with low communication costs.
- **Interaction manager with extensible data types:** Some IMs provide the capability to extend the set of data types used for application/IM communication. This option allows representation conversion to be fully separated from the main body of the application, but it does not do much to solve the semantic feedback problem. Hence it provides only a small increment in flexibility.
- **Extensible interaction manager:** This is accomplished by supplying code that augments or overrides selected internal operations of the IM. An extensible IM can provide as much support as a regular IM for standardized styles of user interface. But it can be used for a wider class of interfaces---including direct manipulation---by taking advantage of its customization capability. This approach provides the most flexibility for meeting user customizability, I/O device variability, and UI style requirements. But it requires substantial processing power, and the level of initial investment (for both UI system development and application developer training) is higher than for any other alternative. Moreover, care is needed to realize the potential flexibility benefits; since application-specific customization code sees a relatively low level of abstraction, it is easy to destroy the logical separation between application and user interface system.

The benefit to be gained from building anything more complex than an abstract device system depends heavily on the degree of standardization of UI behavior---that is, the strength of the UI conventions in the system environment. The more that such conventions limit the range of UI behavior, the more functionality can be put into a toolkit or IM, and the less need there is for an application to override standard behavior. Thus increasing strength of conventions tilts the balance first towards toolkits and extensible IMs, then towards nonextensible IMs.

A nonextensible IM may be the best choice when application portability and development cost are paramount, as it provides the most insulation of the application from UI details. Its limited range of UI styles is a necessary price; at least with present technology, direct manipulation systems cannot be built without significant application involvement in the user interface, which compromises both portability and cost.

B.1.2. Device Interface

The interface between device-independent and device-specific code can be regarded as defining an **abstract device** for the device-independent code to manipulate. The details of an abstract device vary greatly across I/O media, but some general statements can be made about the precision with which the abstract device is specified.

Abstract device variability. This dimension classifies abstract devices according to the degree of variability perceived by the device-independent code.

- **Ideal device:** In this approach, all questions of device variability are hidden from software above the device driver level, so application portability is high. This approach is most useful where the real devices deviate only slightly from the ideal model, or at least not in ways that require rethinking of UI behavior. The ideal-device approach is not appropriate if any major changes in UI behavior may be needed to cope with differences between devices; therefore it cannot cover as wide a range of actual devices as the other two approaches.
- **Parameterized device:** This approach allows a wide range of I/O devices to be accommodated, and it permits substantial changes in UI behavior across devices. The advantage is that application-specific code has both more knowledge of acceptable tradeoffs and more flexibility in changing its behavior than is possible for a device driver. The drawback is that device-independent code may have to perform complex case analysis in order to handle the full range of supported devices. If this must be done in each application, the cost is high and there is a great risk that programmers will omit support for some devices. (To reduce this temptation, it is best to design a parameterized model to have just a few well-defined levels of capability, so as to reduce the number of cases to be considered.) This approach should not be used if it is not necessary to support a wide range of I/O devices, as then its high cost is not repaid. Less obviously, it should not be used when high application portability across I/O devices is crucial (unless the application is insulated from the abstract device by an IM layer); the risk of applications failing to cover the full range of parameter variation is too great. A final drawback is that substantial processing power is likely to be needed to handle extensive runtime case analysis.
- **Device with variable operations:** This approach works best when the device operations are chosen at a level of abstraction high enough to give the device driver considerable freedom of choice. Hence the device-independent code must be willing to give up much control of UI details. This restriction means that direct manipulation (with its heavy dependence on semantically-controlled feedback) is not well supported. Furthermore, only local changes in interface behavior can be handled at the device driver level; changes in basic interface class or application semantics cannot be supported. When these restrictions

are acceptable, this approach can support a very wide range of devices with little impact on device-independent code. Its costs in processing power are low, since runtime case analysis need not be performed.

- **Ad-hoc device:** This approach is hardly ever appropriate for new designs. It is found principally in systems that have evolved from simpler beginnings.

In systems where little or no variation in I/O devices is expected, one may as well specify an ideal device model (tailoring it closely to the real devices); this incurs no runtime cost and provides a well-defined picture of what is required if more devices need to be supported later. When a moderate or wide range of I/O devices must be supported, the key question is what types of UI behavior changes are needed across devices. Parameterization is essential if global changes are needed, as the device driver cannot handle such changes alone. Moderate local changes are well served by the variable-operations method, if its drawbacks are tolerable; otherwise parameterization is preferred. An ideal device approach may still be usable if only small, local changes in behavior are needed.

It is possible to support multiple tradeoffs between handling device adaptation in the device driver and handling it in the application: simple applications can rely on device-specific UI decisions made in the driver, while more complex ones can make their own choices. This amounts to a combination of the variable-operations and parameterized-device approaches. Obviously, to make this work well, great care is needed in defining the device operations and parameters.

Selecting the functions to be provided in an abstract device model is a complex task. A poorly chosen model may limit portability and/or cause performance problems due to mismatches between its properties and specific real devices. Unfortunately, good designs seem very dependent on properties of the particular I/O medium; few general design principles have emerged. We can suggest some rules of thumb based on the chosen degree of variability. When using an ideal or parameterized-device approach, it is probably best to minimize the amount of user interface functionality (i.e., representation conversion, sequence control, user assistance, and state maintenance) placed in the device driver. The variable-operations approach, in contrast, gains its power precisely by moving significant user interface decisions into the device driver. The trick here is to choose a coherent set of decisions that are not tightly coupled to those remaining in higher level software. (Some of the problems with GKS input devices are due to failure to maintain this separation [Rosenthal 81].)

B.2. Representation Issues

After defining the major system components and allocating functionality among them, the next order of business is selecting data representations to be used within the system. We must consider both actual data, in the sense of values passing through the user interface, and *meta-data* that specifies the appearance and behavior of the user interface. Meta-data may exist explicitly in the system (for example, as a data structure describing the layout of a dialog window), or only implicitly. We further subdivide meta-data according to whether it bears on "surface" UI details or on deeper questions of application semantics.

B.2.1. User Interface Definition

Notation for user interface definition. Here we consider the means of defining UI appearance and behavior: the meta-data that describes surface details. We classify notations for UI definition as follows:

- **Implicit in shared user interface code:** This is simple and efficient; it is the appropriate choice for UI behavior that is fixed by the support software. In systems where strong UI conventions exist, quite a lot of the definition can reasonably be represented this way. It should be avoided when the user interface system is to be adaptable across a wide range of UI styles, or when user customizability is important.
- **Implicit in application code:** This is the traditional approach that most UI researchers have tried to move away from. But it will never be eliminated entirely since it, too, is simple and efficient. It is most appropriate where the application is already tightly involved in the user interface, for example, in handling semantic feedback in direct manipulation systems. It should be avoided when application portability (across I/O devices or UI styles) or user customizability is important.
- **External declarative notation:** Declarative representations in general provide the least flexibility of interface design, but are the easiest to use. External declarative notations are particularly well suited to supporting user customization and to use by nonprogramming UI experts. Use of an external notation helps keep the main application code portable across UI styles and I/O devices, but only if the notation is flexible enough to specify all the required variations by itself. Processing power requirements can be high, unless the notation can be precompiled in some way. **Graphical specification** is a special case of external declarative notation; graphical methods are particularly appropriate for specification of visual appearance.
- **Internal declarative notation:** From the application programmer’s viewpoint this is nearly as easy to use as external declarative notation, and it requires much less supporting mechanism; however, it makes user customization much more difficult.
- **External procedural notation:** Procedural notations are more flexible than declarative ones, but are harder to use. User-accessible procedural mechanisms, such as macro definition capability or the programming language of EMACS-like editors, provide very powerful customization possibilities for sophisticated users. Use of an external notation helps keep the main application code portable across UI styles and I/O devices. Substantial processing power may be needed, depending on the efficiency of the mechanism that executes the notation. Also, an external notation by definition has limited access to the state of the application program, which may restrict its capability.
- **Internal procedural notation:** This is the most commonly used notation for customization of extensible interaction managers. It provides an efficient and

flexible notation, but is not accessible to the end user, and so is useless for user customization. It is particularly useful for handling application-specific feedback in direct manipulation interfaces, since it has both adequate flexibility and efficient access to application semantics. This approach is not favored when application portability is a strong requirement.

Typically, several kinds of notation are used in a user interface system. Almost always there are some instances of both kinds of implicit notation, and one or more of the others is often used as well. The crucial question is thus which aspects of UI behavior should be described in which kinds of notation. The best indicators of the appropriate class of notation are the required degrees of flexibility and efficiency.

A good rule of thumb is that declarative notations are appropriate for static information or restricted choices, such as the layout of a display or the selection of one of several predefined behaviors. Procedural notations are a better choice for description of dynamic behavior, because presently available declarative methods aren't sufficiently flexible. In either case, an external notation should be used when user customization is required; otherwise an internal notation is simpler and more efficient. Implicit representation should be used only when efficiency is crucial or the probability of change is low.

B.2.2. Application Semantic Information

Representation of semantic information. This dimension classifies the techniques used for defining application-specific semantic (as opposed to external appearance) information that is needed by the user interface. An example of such information is range restrictions on an input value. The classes are:

- **Implicit.**
- **Declarative.**
- **Procedural.**

The limited range of possibilities allowed by a declarative notation is more of a drawback here than it is for user interface definition. (Semantic information is inherently more variable across applications than surface user interface choices; were this not so, shared UI behavior would be of no interest.) Procedural representations are therefore the best bet where shared code must have access to semantic information, while implicit representations are usually used otherwise. In cases where only a limited number of alternatives are likely to be needed, declarative representations are recommended for ease of use.

Natural language interfaces have special requirements: a great deal of semantic information must be explicitly represented for use in disambiguating sentences. Both declarative and procedural techniques are commonly used.

B.2.3. Representation of Data Values

It turns out that the application interface class is usually sufficient to predict the kinds of data types passed between modules, so the design space does not include a separate dimension for this issue.

The lowest application interface abstraction levels rely on device-related data types, such as bitmaps or other image representations for displays, or keystroke sequences for keyboards. Toolkit systems introduce data types for user interface constructs such as menus or scroll bars. Interaction managers use "internal" data types that might be directly used within application computations, such as integer or floating-point values. Simple IMs use a fixed set of standard internal types, while extensible IMs can be extended to communicate in terms of application-specific internal data types.

As a rule of thumb, application-related data types should be used in preference to device-related data types. For example, integer or Boolean values are preferred to equivalent character strings or bitmaps. This rule encourages moving representation conversions into the user interface code.

B.3. Control Flow and Synchronization

We turn now to questions of control flow: what are the control relationships between the system components, and how are sequences of events synchronized?

It is convenient to visualize control flow in terms of logical *control threads*. A control thread is an entity capable of independently performing computations and waiting for events to occur. We use this term in place of "process" because we do not want to restrict the notion to standard operating-system-supplied processes. (Section B.5.2 lists numerous mechanisms that can support the logical notion of a control thread, possibly with some restrictions in thread structure or event response time.)

Application control flow. Our most basic control flow dimension is a variation of the standard distinction between "internal control" and "external control" [Hayes 85]. We prefer to define the categories as:

- **Single input point.**
- **Multiple input point.**

A single input point is appropriate for creating "modeless" interfaces. Even with a moded interface, building the application in single-input-point style can be helpful, since it serves to decouple the application from details of user interface sequencing. Hence high requirements for application portability or user customizability favor single input point control flow. The major advantage of multiple input point flow is that application actions need not be atomic with respect to user interaction. Generally, multiple input points should be used only if this is an essential feature.

A single input point is also desirable when external events are to be handled while waiting for user input; then there is only one point at which to worry about external events.

Number of control threads. This dimension counts the control threads:

- **Single thread:** This approach is adequate for simple systems, particularly if single input point control flow can be used (i.e., "external control" of the application is sufficient). It is usually not appropriate when external event handling is important, nor when long command execution times occur.
- **One UI thread and one application thread:** This alternative is very popular since it decouples user interface control flow from the application. Two threads are sufficient to allow user interface operations to execute concurrently with the application. On the user interface side, this allows user input to be processed and feedback displays to be updated while commands are being executed. On the application side, external events can be handled without impeding user interface response, and the application is made more independent of user interface event sequencing. The cost of providing a multiple-control-thread mechanism is the major drawback to using this approach. An existing control thread mechanism may be usable, depending on the cost of communication between threads.
- **Multiple UI threads:** Multiple UI threads simplify dealing with logically independent parallel interactions. These occur in modeless interfaces and when multiple input devices are used. An inexpensive thread mechanism is necessary to make this a reasonable approach.
- **Multiple application threads:** Multiple application threads may be useful for dealing with external events. Some systems use them to control cancellation of user commands.

If an inexpensive control thread mechanism is available, the two-thread approach should be used for all but the very simplest user interfaces. The tradeoff point changes if one must build one's own thread mechanism, although a simplified mechanism may be adequate. If independent concurrent sequences of events must be dealt with, explicit use of multiple threads is nearly always the right choice. Even with a restrictive thread mechanism, this will be cleaner and more reliable than ad hoc solutions.

If external event handling is required to preempt user command execution, a thread mechanism that provides preemptive scheduling is very desirable. Without one, it will be necessary to poll for external events during command execution; this is feasible, but inefficient and error-prone.

Treatment of asynchronous input. The user interface must have a strategy for handling asynchronous input events (events that occur while the application is computing). The standard approaches are:

- **Ignore asynchronous input:** Often the simplest approach to implement, and it has some advantages in terms of simplicity of UI behavior. It is usually not appropriate if commands may take a long time to execute.

- **Queue before all processing:** A satisfactory solution if events do not remain in the queue for long. Otherwise, the lack of feedback is a serious human factors shortcoming. Hence this approach is also inappropriate if commands may take a long time to execute; but it is usually the best solution for short or intermediate command times.
- **Partial processing with queuing:** Provides flexibility, but requires multiple control threads and introduces synchronization concerns. Hence it should be avoided unless necessary (i.e., unless there are long commands).

Fast input processing. When partial processing is provided, the variability of behavior of the fast processing is an important issue. This may be:

- **Fixed behavior:** Simple and has no synchronization problems, but is obviously inflexible. It is sufficient if user interface system adaptability is not a strong requirement.
- **Parameterized behavior:** Recommended in most cases, because the parameter semantics can be defined to minimize synchronization problems. (In particular, one should be wary of parameters that will be changed "on the fly" when already-processed input may be pending.)
- **Application-dependent behavior:** Should be used only if user interface system adaptability requirements are so high as to make it mandatory. Use of application-supplied fast processing routines reduces application portability and creates significant synchronization concerns.

The more flexible alternatives in this dimension carry increasing risk of synchronization problems. (A simple example is that typed-ahead characters may be echoed twice or not at all when switching between asynchronous echoing and application-driven echoing.) Communication costs can also be a problem for the last alternative. In general, one should use the least flexible method possible.

Application communication grain size. How frequently does communication occur between application and shared user interface code?

- **Coarse grain:** This is suitable when the application need not be involved in the details of UI interactions.
- **Fine grain:** This is most likely to be required in direct manipulation interfaces. Communication costs and application portability are sacrificed, so this alternative should not be used unless necessary.

Coarse-grained communication should be used if the application has long-running commands or external events to cope with, since then one cannot rely on it to provide feedback promptly.

It is also possible to distinguish between coarse-grained and fine-grained communication at the device interface. In coarse-grained device communication, the device-specific code handles feedback for entire sequences of input events; while with fine-grained device communication, feedback is handled at higher levels. As a rule of thumb, fine-grained device communication is preferable. Coarse-grained communication may be acceptable if substantial control of the user interface is to be put in the device driver level; this is associated with the device interface classification of abstract devices with variable operations.

B.4. Matters of State

The system architecture should explicitly recognize state information, whether hidden within one module or shared between modules. Shared state is a useful vehicle for communication. Shared or not, the existence of persistent state is a key aspect of system semantics and an important basis for performance optimization.

B.4.1. Representation of Interface State

How to represent the state of the user interface is a very general question. Our rules of thumb address only a small part of it, to wit: whether to retain intermediate representations of output (such as display lists or cached bitmaps). Intermediate representations take extra work to maintain, but can provide valuable benefits. We recommend maintaining an intermediate output representation when (1) the output device can usefully be treated as having a state (not true for audio output, for instance); (2) recalculating the output device's state from scratch (from underlying application state) is expensive; and (3) partial or incremental updates are common. Under these conditions the performance gain is worth the extra trouble.

Intermediate output representations are also important for handling reference interpretation (e.g., deducing that a mouse click represents a menu element selection). This may justify maintaining an intermediate representation even when display update savings are not significant. A partial representation (e.g., just menu coordinates) may be enough for this purpose.

B.4.2. Communication via Shared State

Basis of communication. Communication between modules may depend on shared state or on events, or both. (An *event* is a transfer of information occuring at a discrete time, for example via a procedure call or message. Communication through shared state variables is significantly different because the recipient need not use information in the same order in which it is sent.)

- **Events.**
- **Pure state.**
- **State with hints.**
- **State plus events.**

State-based communication can be recommended for driving devices that exhibit persistent state, such as displays. The use of explicit state is a natural way of formalizing the maintenance of intermediate representations of output (see above). However, event-based communication is more appropriate for devices that have no useful characterization of state.

The hybrid communication forms which combine events with shared state allow improved performance at the price of increased complexity. As a rule of thumb, pure state systems are simpler and less efficient than pure event systems, which in turn are simpler and less efficient than hybrid systems.

The major drawback to state-based communication is that it requires efficient access to shared storage. This may not be available in multi-process systems, especially when communication across network links is involved. Synchronization issues must also be considered if multiple threads access the shared state.

B.5. Mechanisms

The final group of dimensions concern the mechanisms used to implement communication and control flow. The classifications used here are the lowest level of detail that can reasonably be described as part of the system architecture. But these issues are indeed part of system architecture, because they have strong implications for questions that we have already discussed.

B.5.1. Communication Mechanisms

Event mechanisms. A pure state-based system has no events and so needs no event communication mechanism. The other three classes of communication require a mechanism to pass events between modules. For communication within a single control thread, the alternatives are:

- **Direct procedure call.**
- **Indirect procedure call.**

Indirect calls provide useful separation between the communicating modules. If the chosen programming language has a natural mechanism for representing indirect calls, they are usually well worth the small runtime cost; but otherwise the difficulty of using indirect calls may outweigh their value.

For communication between control threads, the alternatives are:

- **Asynchronous message.**
- **Synchronous message.**

Asynchronous messages are often superior since they reduce synchronization problems and can be batched to reduce overhead. Synchronous messages have simpler semantics and sometimes can be implemented more easily (e.g., message buffers may not be needed). If a message mechanism is already available, one should probably use it by default; otherwise asynchronous messages seem better suited to most UI purposes.

Separation mechanisms. A separation mechanism isolates software components while still permitting communication. We recognize four classes:

- **Programming convention:** This approach provides very weak protection, but it is flexible and incurs no runtime cost. This is a reasonable choice for communication between closely related components, or when the system components are automatically generated (and thus less prone to human coding error).
- **Visibility rules:** This type of mechanism is quite flexible, since the programmer can choose what to export or hide. The runtime cost is small: at most, a procedure call is needed to cross a protection boundary. In many programming languages the protection is not secure against runtime errors.
- **Hardware separation:** Security is strong, but the cost of communicating across the protection boundary is high---often several orders of magnitude more expensive than a procedure call. This is an appropriate choice when it is important to ensure security, for example in a window manager that serves multiple applications. This approach may also be necessary for communication between modules coded in different programming languages. An important aspect of hardware separation is that most current operating systems associate these protection boundaries with processes; hence division of the user interface system into protectable entities must be considered jointly with control flow and synchronization concerns.
- **Network link:** The communicating parties can exist on nonidentical machines. The cost of communication in such a case is inherently high, but is worthwhile in distributed environments.

Generally, visibility rules are the minimum separation that should be used between application and user interface code. Stronger separation mechanisms should be used only where there are system considerations that justify their cost. The major considerations that may justify a stronger mechanism are (1) the need for a shared user interface system to protect itself against errors in any one application; (2) use of a system-provided process mechanism that forces hardware separation; or (3) the desire to distribute system components across machines in a network.

Separation will also exist between the shared user interface code and the device-specific code. This may or may not use the same class of mechanism as is used at the application interface. In most cases visibility rules are sufficient; the main exception is to permit distribution across a network.

B.5.2. Control Flow Mechanisms

Control thread mechanism. Among the many ways to provide the abstract notion of a control thread are:

- **Standard processes:** These provide security against other processes, but interprocess communication is relatively expensive. For a user interface system, security may or may not be a concern, while communication costs are almost always a major concern. If the operating system already provides processes, not having to implement one's own process mechanism is an important advantage. In network environments, standard processes are usually the only kind that can be executed on different machines.
- **Lightweight processes:** These are suitable only for mutually trusting processes due to lack of security; but often that is not a problem for user interface systems. The benefit is substantially reduced cost of communication, especially for use of shared variables. Few operating systems provide lightweight processes, and building one's own lightweight process mechanism can be difficult.
- **Non-preemptive processes:** These are relatively simple to implement since no preemption mechanism is needed. Synchronization can be achieved merely by not yielding the processor, although explicit interlocks are safer. The major drawback is that response to I/O devices can be slow, and response time is hard to control.
- **Interrupt service routines:** These provide a simple preemptive scheduling mechanism. The control flow and communication patterns of ISR-implemented processes are very restricted, but they are useful for ensuring fast response to I/O devices. ISRs are highly machine-dependent, and may not be available to unprivileged programs.
- **Event handlers:** The main advantage of this method is that it requires virtually no support mechanism. The key disadvantages are the control flow restrictions, which are comparable to ISRs, and the lack of fast response, which is comparable to non-preemptive processes.

Of these, the most commonly useful alternatives are standard processes, lightweight processes, and event handlers; the others are appropriate only in special cases. For most user interface work, lightweight processes are very appropriate if available. Standard processes should be used when protection considerations warrant, and in network environments where it may be useful to put the processes on separate machines. If these conditions do not apply, event handlers are the best choice when their response time limitations are acceptable; otherwise it is probably best to invest in building a lightweight process mechanism.

References

[Adobe 85]	*PostScript Language Reference Manual* Adobe Systems, Inc., 1985. Published by Addison-Wesley.
[Apple 85]	*Inside Macintosh* Apple Computer, Inc., 1985. Published by Addison-Wesley. Volumes I-IV.
[Bell 71]	C. Gordon Bell and Allen Newell. *Computer Structures: Readings and Examples.* McGraw-Hill, New York, 1971.
[Borenstein 88]	Nathaniel S. Borenstein and James Gosling. UNIX Emacs: A Retrospective (Lessons for Flexible System Design). In *Proceedings of Symposium on User Interface Software*, pages 95-101. ACM SIGGRAPH, Banff, Alberta, Canada, October 1988.
[Dance 87]	John R. Dance, Tamar E. Granor, Ralph D. Hill, Scott E. Hudson, Jon Meads, Brad A. Myers, and Andrew Schulert. The Run-time Structure of UIMS-Supported Applications. *Computer Graphics* 21(2):97-101, April 1987. ACM SIGGRAPH Workshop on Software Tools for User Interface Management.
[Green 86]	Mark Green. A Survey of Three Dialogue Models. *ACM Transactions on Graphics* 5(3):244-275, July 1986.
[Hartson 89]	H. Rex Hartson and Deborah Hix. Human-Computer Interface Development: Concepts and Systems for Its Management. *ACM Computing Surveys* 21(1):5-92, March 1989.
[Hayes 85]	Philip J. Hayes, Pedro A. Szekely, and Richard A. Lerner. Design Alternatives for User Interface Management Systems Based on Experience with Cousin. In *Proceedings of CHI '85: Human Factors in Computing Systems*, pages 169-175. ACM SIGCHI, 1985.
[Knuth 73]	Donald E. Knuth. *The Art of Computer Programming*. Volume 1: *Fundamental Algorithms.* Addison-Wesley, Reading, MA, 1973.
[Lane 90a]	Thomas G. Lane. *User Interface Software Structures*. PhD thesis, Carnegie Mellon University, May 1990. CMU School of Computer Science Technical Report CMU-CS-90-101. Also available as Software Engineering Institute Special Report CMU/SEI-90-SR-13.

[Lane 90b] Thomas G. Lane.
Studying Software Architecture through Design Spaces and Rules.
Technical Report CMU/SEI-90-TR-18, Carnegie Mellon University Software Engineering Institute, October 1990.
Also available as CMU School of Computer Science Technical Report CMU-CS-90-175.

[Lantz 87] Keith A. Lantz, Peter P. Tanner, Carl Binding, Kuan-Tsae Huang, and Andrew Dwelly.
Reference Models, Window Systems, and Concurrency.
Computer Graphics 21(2):87-97, April 1987.
ACM SIGGRAPH Workshop on Software Tools for User Interface Management.

[Myers 89] Brad A. Myers.
User-Interface Tools: Introduction and Survey.
IEEE Software 6(1):15-23, January 1989.
An earlier version was published as Carnegie Mellon University Technical Report CMU-CS-88-107, January, 1988.

[Perry 84] Robert H. Perry, Don W. Green, and James O. Maloney.
Perry's Chemical Engineers' Handbook.
McGraw-Hill, New York, 1984.

[Reid 80] Brian K. Reid and Janet H. Walker.
Scribe User's Manual
Third edition, Unilogic, Ltd., May 1980.

[Rosenthal 81] David S. H. Rosenthal.
Methodology in Computer Graphics Re-examined.
Computer Graphics 15(2):152-162, July 1981.

[Rosenthal 82] David S. H. Rosenthal, James C. Michener, Gunther Pfaff, Rens Kessener, and Malcolm Sabin.
The Detailed Semantics of Graphics Input Devices.
Computer Graphics 16(3):33-38, July 1982.

[Scheifler 86] Robert W. Scheifler and Jim Gettys.
The X Window System.
ACM Transactions on Graphics 5(2):79-109, April 1986.

[Sedgewick 88] Robert Sedgewick.
Algorithms.
Addison-Wesley, Reading, MA, 1988.

[Shaw 86] Mary Shaw.
An Input-Output Model for Interactive Systems.
In *Proceedings of CHI '86: Human Factors in Computing Systems*, pages 261-273. ACM SIGCHI, 1986.

[Shaw 89] Mary Shaw.
Larger Scale Systems Require Higher-Level Abstractions.
In *Proceedings of Fifth International Workshop on Software Specification and Design*, pages 143-146. IEEE Computer Society, May 1989.
ACM SIGSOFT Software Engineering Notes, Volume 14 Number 3.

[Shneiderman 86] Ben Shneiderman.
Seven Plus or Minus Two Central Issues in Human-Computer Interaction.
In *Proceedings of CHI '86: Human Factors in Computing Systems*, pages 343-349. ACM SIGCHI, 1986.

[Tanner 83] Peter Tanner and William Buxton.
Some Issues in Future User Interface Management System Development.
In Gunther Pfaff (editor), *Seeheim Workshop on User Interface Management Systems*, pages 67-79. EUROGRAPHICS-Springer, 1983.

[Wegner 87] Peter Wegner.
Dimensions of Object-Based Language Design.
Sigplan Notices 22(12):168-182, December 1987.
Proceedings of OOPSLA '87: Conference on Object-Oriented Programming Systems, Languages, and Applications.

Table of Contents

List of Figures