NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

Studying Software Architecture through Design Spaces and Rules

Thomas G. Lane

October 1990
CMU-CS-90-175$_2$

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

This report has also appeared as Carnegie Mellon University
Software Engineering Institute Technical Report No. CMU/SEI-90-TR-18.

Abstract

This report argues that the overall structure of software systems ("software architecture") is usefully studied by constructing design spaces. A design space identifies the key functional and structural choices made in creating a system design, and it classifies the alternatives available for each choice. Rules can be formulated to relate choices within a design space. Sets of such rules are a valuable design aid and offer a promising route to automatic structural design. By codifying design practice, design spaces can also aid software maintenance and training. To support this argument, the report describes a design space and associated rules for user interface software, and it discusses an experiment that validated these design rules by comparing their predictions to real system designs.

This research was sponsored in part by the Defense Advanced Research Projects Agency (DoD), monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, Ohio 45433-6543, under Contract F33615-87-C-1499, ARPA Order No. 4976, Amendment 20; and in part by the U.S. Department of Defense under Contract F19628-90-C-0003, with the U.S. Air Force as the executive contracting agent.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring agencies or of the U.S. Government.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Keywords: Software engineering, design methodologies, design representations, software architecture, user interfaces.

Table of Contents

1. Introduction
   1.1 The Utility of Codified Knowledge
   1.2 The Notion of a Design Space
   1.3 Related Work

2. A Design Space for User Interface Architectures
   2.1 A Basic Structural Model
   2.2 Sample Functional Dimensions
      2.2.1 External Requirements
      2.2.2 Basic Interactive Behavior
      2.2.3 Practical Considerations
   2.3 Sample Structural Dimensions
      2.3.1 Division of Functions and Knowledge Between Modules
      2.3.2 Representation Issues
      2.3.3 Control Flow, Communication, and Synchronization Issues

3. Design Rules for User Interface Architecture
   3.1 Sample Rules

4. Applying the Design Space: An Example

5. A Validation Experiment

6. How the Design Space Was Prepared

7. Summary

Acknowledgements

References

List of Figures

Figure 1-1: A Simple Design Space
Figure 2-1: A Basic Structural Model for User Interface Software
Figure 2-2: The Sample Design Space Dimensions

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.

The architectural alternatives available to a system designer can be described and classified by constructing a *design space*. Within a design space, we can formulate design rules that indicate good and bad combinations of choices. Such rules can be used to select an appropriate system design based on functional requirements. The design space is useful in its own right as a shared vocabulary for describing and understanding systems.

This work should be viewed as a means of codifying software design knowledge for use in day-to-day practice and in the training of new software engineers. For this purpose, a set of design rules need not produce a "perfect" or "best possible" design. A valuable contribution will be made if the rules can help a journeyman designer to make choices comparable to those that a master designer would make---or even just help the journeyman to choose a reasonable design with no major errors. With sufficient experience, a set of such rules may become complete and reliable enough to serve as the basis for automated system design, but the rules can be of practical use long before that stage is reached.

The work described in this report tested these notions by constructing a design space and rules for the architecture of user interface software systems. These rules were experimentally tested by comparing their recommendations to actual system designs. The results showed that a rather simple set of rules could achieve a promising degree of agreement with the choices of expert designers. These exploratory results suggest that the approach sketched here is a viable means of creating an organized body of knowledge for software engineering.

This report is a summary of results from the author's thesis [Lane 90a]. A companion report presents the user interface design space and rules in greater detail [Lane 90b].

1.1 The Utility of Codified Knowledge

The underlying goal of this work is to organize and express software design knowledge in a useful form. One way of doing this is to build up a vocabulary of well-understood, reusable design concepts and patterns. If widely adopted, a design vocabulary has three major benefits. First, it aids in creating a system design by providing mental building blocks. Second, it helps in understanding or predicting the properties of a design by offering a context for the creation and application of knowledge. Third, it reduces the effort needed to understand another person's design by reducing the number of new concepts to be learned.

An example of such a vocabulary is the codification of control structures that took place about two decades ago. Programmers learned to perceive control flow in terms of a few standard concepts (conditionals, iteration, selection, subroutine calls, etc.) rather than as a complex pattern of low-level tests and branches. By reducing apparent complexity and providing a shared understanding of control flow patterns, use of these building blocks made programs both easier to write and easier to read. Researchers discovered key properties of these structures, for example, the invariant and termination conditions of loops. Use of the standard structures helped practitioners to focus on these properties, leading to better-understood, more reliable programs. Finally, codification made it possible to build tools (programming languages) that supported the structural concepts directly, providing further productivity gains.

As software engineering matures and research attention shifts to ever-larger problems, we can expect to see similar codification occurring for larger software entities. The time now seems ripe to begin codifying structural patterns in medium-size software systems, to wit, characteristics of modules and the interconnections between them. (We can already anticipate that even higher levels of design abstraction will be needed to design very large systems, but we are far from having enough experience to be able to discern patterns at that scale.)

A different analogy for this work is the compilation of engineering design handbooks, such as [Perry 84]. The established fields of engineering have long distinguished between innovative and routine design. Innovative design relies upon raw invention or derivation from abstract principles, while routine design uses standardized methods to solve problems similar to those that have been solved before. When applicable, routine design methods are cheaper and more likely to yield an acceptable (though not necessarily optimum) design than are innovative methods. The primary purpose of such handbooks is 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. Thus, software engineering handbooks could combat two opposite evils now widely seen in practice: both the tendency to invent every new system from scratch and the tendency to reuse a single design for every problem regardless of its suitability. Handbook-style texts are now widely available for selection of algorithms and data structures (e.g., [Knuth 73, Sedgewick 88]) but do not yet exist for higher levels of software design.

The work reported here offers an organizational scheme (namely, design spaces and rules) for handbooks of software system structure, as well as the beginnings of specific knowledge for one such handbook (covering user interface 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.

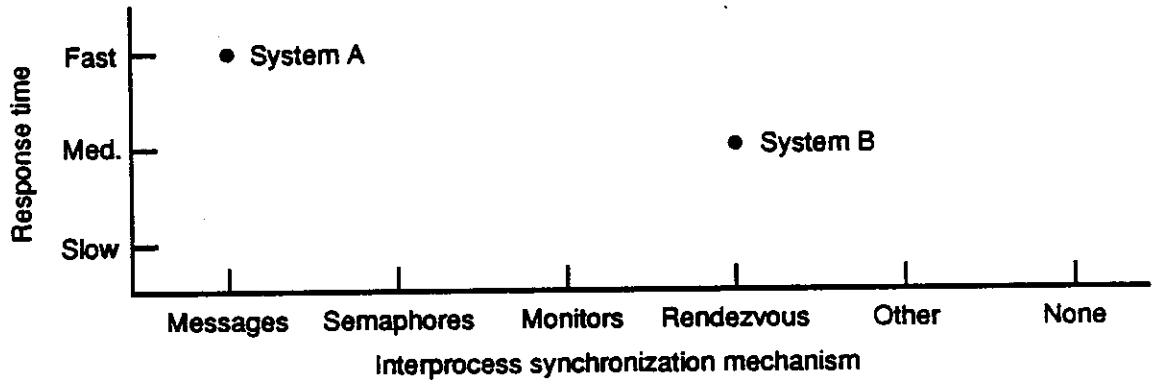

Figure 1-1: A Simple Design Space

The different dimensions are not necessarily independent; in fact, it is important to discover correlations between 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 key part of the design space approach is to choose some dimensions that reflect requirements or evaluation criteria (function and/or performance), while other dimensions reflect structure (or other available design choices). Then, any correlations found between these dimensions can provide direct design guidance: they show which design choices are most likely to meet the functional requirements for a new system. For example, the hypothetical data in Figure 1-1 suggest that a message mechanism is more likely to provide fast response time than a rendezvous mechanism. (Of course, one would want more than just two data points before drawing this conclusion.)

The dimensions that describe functional and performance requirements make up the *functional design space*, while those that describe structural choices make up the *structural design space*. These groupings can be regarded either as independent spaces or as subspaces of a single large design space. In the context of a stepwise ("waterfall") model of the software design process, the functional design space represents the results of the requirements analysis and gross functional design steps, while the structural design space represents the results of initial system decomposition.

The dimensions of a design space are usually not continuous and need not possess any useful metric (distance measure). A dimension that represents a structural choice is likely to have a discrete set of possible values, which may or may not have any meaningful ordering. 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 (e.g., a performance number), one may choose to aggregate it into a few discrete values (e.g., "low," "medium," "high"). This is appropriate when such gross estimates provide as much information as one needs or can get, as is often true in the early stages of design.

1.3 Related Work

A seminal use of the design space concept 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]. Design-space-like schemes have also been proposed by workers in software reuse; for example, Prieto-Diaz and Freeman index a software library using sets of "terms" (keywords) grouped into "facets" [Prieto 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. A Design Space for User Interface Architectures

The design space reported here, together with its associated rules, describes architectural alternatives for user interface software: systems whose main focus is on providing an interactive user interface for some software function(s). The system studied need not provide the whole user interface. Thus the scope of the study included not only complete user interface management systems (UIMSs), but also graphics packages, user interface toolkits, window managers, and even standalone applications that have a large user interface component. This scope is large enough to include a wide range of useful system structures, yet not so large as to be intractable. While another domain could have been chosen, user interfaces are a good choice because the field is in ferment, with little agreement on the best possible structures. Hence the results may be useful immediately, in addition to serving to illustrate the larger argument made above.

The design space is too large to cover completely in this report. Therefore only some representative dimensions and rules will be described. (For a more complete presentation of the space, see [Lane 90b].) The complete design space contains 25 functional dimensions, 6 of which are described here. Three to five alternatives are recognized in each of these dimensions. There are 19 structural dimensions (5 of which are described here), each offering two to seven alternatives. Figure 2-2 presents the dimensions discussed in this report.

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. This consists of 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. This consists of 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. This consists of 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).

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, regarding the server as a device-specific black box.

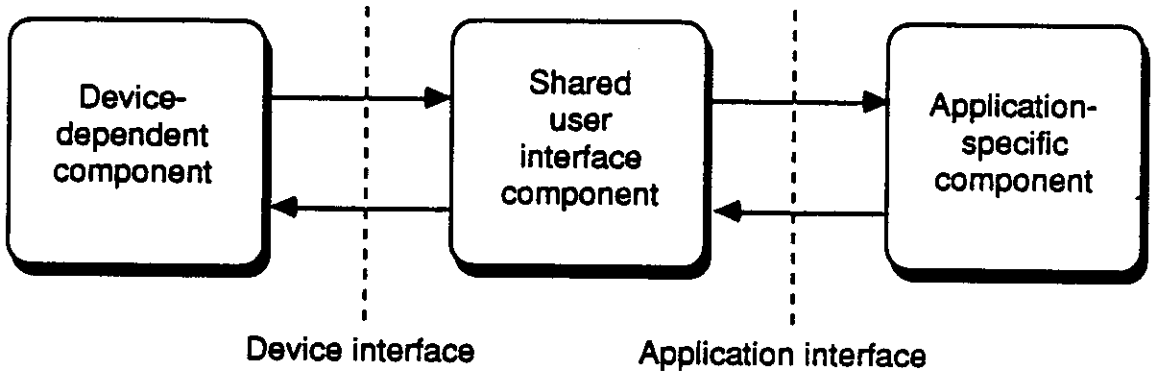

Figure 2-1: A Basic Structural Model for User Interface Software

2.2 Sample Functional Dimensions

The functional dimensions identify the requirements for a user interface system that most affect its structure. These dimensions fall into three groups:

- **External requirements.** This group 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.** This group includes the key decisions about user interface behavior that fundamentally influence internal structure.
- **Practical considerations.** This group covers development cost considerations; primarily, the required degree of adaptability of the system.

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.

2.2.1 External Requirements

External event handling is an example of a dimension reflecting an application-imposed external requirement. This dimension indicates whether the application program needs to respond to external events (defined as events not originating in the user interface), and if so, on what time scale. The design space recognizes three alternative choices:

- **No external events:** the application is not influenced 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. Automatic reporting of mail arrival might be handled this way.
- **External events preempt user commands:** external event servicing has sufficiently high priority that user command execution must be interrupted when an external event occurs. This requirement is common in real-time control systems.

User customizability is an example of a user-imposed external requirement. The design space recognizes three levels of end user customizability of a user interface:

Functional Dimensions

External event handling
- No external events
- Process events while waiting for input
- External events preempt user commands

User customizability
- High
- Medium
- Low

User interface adaptability across devices
- None
- Local behavior changes
- Global behavior changes
- Application semantics changes

Computer system organization
- Uniprocessing
- Multiprocessing
- Distributed processing

Basic interface class
- Menu selection
- Form filling
- Command language
- Natural language
- Direct manipulation

Application portability across user interface styles
- High
- Medium
- Low

Structural Dimensions

Application interface abstraction level
- Monolithic program
- Abstract device
- Toolkit
- Interaction manager with fixed data types
- Interaction manager with extensible data types
- Extensible interaction manager

Abstract device variability
- Ideal device
- Parameterized device
- Device with variable operations
- Ad-hoc device

Notation for user interface definition
- Implicit in shared user interface code
- Implicit in application code
- External declarative notation
- External procedural notation
- Internal declarative notation
- Internal procedural notation

Basis of communication
- Events
- Pure state
- State with hints
- State plus events

Control thread mechanism
- None
- Standard processes
- Lightweight processes
- Non-preemptive processes
- Event handlers
- Interrupt service routines

Figure 2-2: The Sample Design Space Dimensions

- **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, window sizes, colors, etc.
- **Low:** little or no user customizability is required.

User interface adaptability across devices depends on the expected range of I/O devices that the user interface system must support. This dimension indicates the extent of change in user interface behavior that 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 occur across devices, for example, in the appearance of menus.
- **Global behavior changes:** there are major changes in surface user interface behavior across devices, for example, a change in basic interface class (see below).
- **Application semantics changes:** there are changes in underlying semantics of commands (e.g., continuous display of state versus display on command).

Computer system organization is an example of a dimension describing the surrounding computer system. This dimension classifies the basic nature of the environment as follows:

- **Uniprocessing:** only one application executes at a time.
- **Multiprocessing:** multiple applications execute concurrently.
- **Distributed processing:** environment is a computer network, with multiple CPUs and non-negligible communication costs.

2.2.2 Basic Interactive Behavior

Basic interface class 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. Resolution of ambiguous input is a key problem.
- **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.

2.2.3 Practical Considerations

Application portability across user interface styles is an example of a dimension defining the required degree of adaptability of a user interface system. This dimension specifies the degree to which application-specific code is insulated from user interface style changes.

- **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.

2.3 Sample Structural Dimensions

The structural dimensions represent the decisions determining the overall structure of a user interface system. These dimensions also fall into three major groups:

- **Division of functions and knowledge between modules.** This group considers how system functions are divided into modules, the interfaces between modules, and the information contained within each module.
- **Representation issues.** This group considers 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 dialogue window), or only implicitly.
- **Control flow, communication, and synchronization issues.** This group considers the dynamic behavior of the user interface code.

2.3.1 Division of Functions and Knowledge Between Modules

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 such 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

[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 is based on the theoretical distinctions made by Hayes.

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.

- **Interaction manager with extensible data types:** similar to the previous category, except that 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:** again, communication between the application and shared code is 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 there is a significant body of application-specific code that customizes the interaction manager; this code is much more tightly coupled to the internal details of the interaction manager than is the case for 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 are 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.

---

[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].

- **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. Examples are 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, an abstract device with variable operations is useful when much of the system's "intelligence" is to be put into the device-specific layer; but it is only appropriate for handling local changes in user interface behavior across devices.

2.3.2 Representation Issues

Notation for user interface definition is a representation dimension. It 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 non-procedural specification separate from the body of the application program, for example, a grammar or tabular specification. External declarative notations are particularly well suited for supporting user customization and for use by non-programming 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 definition 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 non-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. 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 notation is the easiest to use (especially for non-programming user interface designers) but it has the least power, since it can only represent a predetermined range of possibilities. Typically, several notational techniques are used in a system, with different aspects of the user interface being 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.

2.3.3 Control Flow, Communication, and Synchronization Issues

Basis of communication is a communication dimension. This dimension classifies systems according to whether communication between modules depends upon shared state, 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 classification is:

- **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.

Control thread mechanism describes the method, if any, used to support multiple logical threads of control. Multiple threads are extremely useful in user interface systems, for example in handling multiple input devices or for decoupling application processing from user interface logic. 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 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 that is often a nonissue 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, for example, 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.

3. Design Rules for User Interface Architecture

There are very few hard-and-fast rules at this level of design. Most connections between design dimensions are better described by saying that a given choice along one dimension favors or disfavors particular choices along another dimension; the strength of this correlation varies from case to case. The designer's task is to consider all such correlations and to select the alternative favored by the preponderance of the evidence.

Therefore, a natural notation for a design rule is a positive or negative weight associated with particular combinations of alternatives from two (or more) dimensions. A given design can be evaluated by summing the weights of all applicable rules. The "best" design is then the one with the highest score. The author prepared a mechanically evaluatable set of design rules of this form and an evaluation program that would rank the structural alternatives when given a set of values for the functional dimensions. (Section 5 describes an experimental test of this rule set.) The rules can also be viewed less formally as guidelines for human designers.

It is useful to distinguish two categories of rules: those linking functional to structural dimensions, and those interconnecting structural dimensions. The first group allows system requirements to drive a structural design, while the second group ensures the internal consistency of the design.[3] This second group complicates the task of finding the design with the highest score, since choices in different dimensions affect each other. The author resorted to combinatorial searching to locate the best designs; better algorithms may be found in the future. A possible source of better methods is "neural network" techniques, which seem to have some similarity to this problem.

The mechanical design rule set contains 622 rules; these rules are written in a very primitive notation and can be reduced to about 170 rules at a more reasonable level of abstraction. The very abbreviated descriptions below account for about ten percent of the formal rules.

3.1 Sample Rules

The earlier descriptions of structural alternatives already mentioned some of the conditions under which one alternative may be preferred to another. This section presents more formally some of the specific design rules that connect the sample dimensions. Each of the sample rules is given in prose form, together with a brief justification.

- If external event handling requires preemption of user commands, then 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.
- 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 external notations.
- 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.
- A distributed system organization favors event-based communication. State-based communication requires shared memory or some equivalent, which is often expensive to access in such an environment.
- The basic user interface class affects the best choice of application interface abstraction level. For example, menu selection and form filling user interfaces are well served by toolkits and nonextensible interaction managers. But experience has shown that nonextensible interaction managers are not adequate for direct manipulation, because they don't handle semantic feedback well. Extensible interaction managers and toolkits are the favored alternatives for direct manipulation.
- 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.

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.

4. Applying the Design Space: An Example

To illustrate these ideas, this section presents a concrete example. The sample system is the cT programming language and environment [Sherwood 88]. cT is designed for the creation of high-quality, interactive educational applications, for example, physics simulations or instruction in musical notation. It must be usable by authors who are experts in their particular subject matter, but who have only limited programming experience. cT implementations exist on a variety of personal computers and workstations, and portability of application programs across these platforms is an important goal.

cT's functional requirements can be described in the terms of the design space. For the sample dimensions previously cited:

- There is no requirement for external event handling; it's not needed in the target class of applications.
- Little or no end user customizability is needed.
- User interface adaptability across devices may require local behavior changes, for instance to fill areas with different patterns when color is not available. The range of supported platforms is not so wide that global behavior changes might be necessary.
- Computer system organization may be uniprocessing or multiprocessing. cT does not make special provisions for distributed systems.
- Basic interface class is usually direct manipulation, but menu selection is also used. Each application determines its basic interactive behavior.
- Medium portability of applications across user interface styles is required. In such things as menu appearance, cT follows the conventions of the host platform, and the application should be independent of such details.

To describe cT structurally, we classify the cT programming system itself as the shared user interface code, instructional programs written in cT as application-specific code, and the underlying platform (including graphics packages, etc.) as device-specific code. (Notice that this division is already implicit in the functional classification above.)

The architecture of cT can then be classified in the sample structural dimensions as follows:

- The application interface abstraction level falls in the toolkit class. Toolkit elements are provided for common constructs such as menus or scrolling text boxes. cT's toolbox is particularly strong in the analysis of text input (recognition of misspelled words, equivalent forms of algebraic expressions, etc). For other interactive behavior the application resorts to manipulation of the underlying abstract device.
- The device interface uses a parameterized abstract device. Decisions such as how to scale displays to fit the available hardware are handled largely by the shared user interface code (but the application can set policy, such as whether to preserve aspect ratio).
- User interface notation is mostly implicit; some aspects are implicit in the shared code while others are implicit in the application. Limited use is made of internal procedural notation, and there are some toolbox parameters that qualify as internal declarative notation.
- Communication is based on events; no shared state variables are used.
- cT uses basically a single thread of execution. An exception occurs in the development environment: while editing a cT program, incremental recompilation is done while waiting for user input. The "background" control thread used for this purpose is implemented with an event handler mechanism.

The mechanical rule set is largely able to replicate these design decisions. For example, the rules recommend implicit and internal-procedural user interface notations, because the requirements for user customizability and application portability are not high enough to justify the extra cost of external or declarative notations.[4] The rules recommend strict single-thread control flow, so they disagree on the last of the sample dimensions. This is unsurprising since the decision to provide background recompilation is outside the scope of the present design space.

[4]Other functional dimensions, not discussed in this report, also enter into this conclusion.

5. A Validation Experiment

To test the validity of the design space and rules, the rules' recommendations were compared to the actual designs of some user interface systems. This experiment used six systems that had not been studied in the course of preparing the design space and rules. The test was carried out as follows:

- A designer of each system was asked to describe his system in the terms of the design space; that is, to choose the most descriptive category in each functional and structural dimension.
- Each system's functional description was fed into a program that searched for the structural alternatives that were most highly rated by the rule set.
- The resulting structural recommendations were compared to the actual system descriptions.

The six systems covered a fairly wide range of user interface requirements. Among them were two radically different UIMSs, an integrated programming environment for teaching novice programmers, the cT system described above, a system for automatic creation of graphical database displays, and a flight simulator control program. Most of the systems have seen extensive use, so the designers' functional descriptions generally reflect actual experience rather than goals or guesses.

The test showed a moderate to substantial degree of agreement between the rules' predictions and the actual system designs, according to the standard interpretation of the kappa statistic [Landis 77]. Most of the discrepancies could be classified either as legitimate differences of design opinion, or as small errors or oversights in the rules that had not come to light in prior test cases. (An example of such an error is that the rules treat all varieties of state-based communication as about equally expensive in processing power, whereas actually the forms providing hints are more efficient than pure-state communication. To maintain experimental rigor, the rules were not modified to correct such errors after the formal experiment began.)

The only area in which the rules showed little correlation to the actual designs was that of representational choices: the notation for user interface definition dimension described previously, and one other dimension that provides a similar classification for representation of application-specific semantic information. It may be that corrections and additions to the rules would improve this result. However, both of these design choices are heavily influenced by considerations of design-time methods and procedure. Since the present design space deals mainly with issues of run-time structure, it may well be that the space provides insufficient information to make correct choices in these two dimensions. In that case, the space would need to be extended to cover questions of design procedure before these dimensions could be handled reliably.

These are remarkably good results when one considers the limited amount of information in the rules (Section 3.1 alone contains about ten percent of the full set). This suggests that the design space provides considerable leverage for the rules; that is, that the classifications made by the design space make it easier to select the right type of design.

Furthermore, these rules were developed and tuned to follow the author's own judgments and those of the designers whose systems he studied while preparing the design space. This experiment compared the rules to the judgments of a completely separate group of designers. The extent of correlation is therefore especially striking: it depends not only on the rules successfully representing the knowledge on which they were based, but on agreement between two unrelated groups of designers. Therefore, this experiment shows that:

- There is a significant body of agreement among expert user interface system designers about structural choices.
- The design space and rules capture (at least part of) that agreement.

Though the particular set of rules tested in this experiment possess numerous faults, these results strongly suggest that the overall approach is valid and powerful.

This experiment is described in more detail in [Lane 90a].

6. How the Design Space Was Prepared

The design space and rules described here were based on an extensive survey of existing user interface systems. 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.

An obvious limitation of this approach is that it may not result in much insight about new, never-before-seen structures (although the design space can call attention to untried combinations of known alternatives). However, for the purpose of codifying known practice this is not a major problem.

A more serious objection is that important dimensions may be overlooked. It seems very difficult to demonstrate that a given design space covers everything that may be of interest at a particular level of abstraction. (Obviously a practical design space cannot cover all possible ways of looking at a software system, so some such restriction is necessary.) The experimental results suggest that functional (and perhaps also structural) dimensions associated with design methods may need to be added to the present space, so it is clear that the risk is real.

In the other direction, experience with this space suggests that getting rid of extraneous dimensions is just as important and difficult. For example, of the twenty-five functional dimensions originally defined for the space, it turns out that only about ten or twelve have significant impact; the other dozen seem to have considerably less influence, and perhaps should have been omitted entirely. In the structural dimensions, it proved possible to omit many design choices because they turned out to be closely correlated with choices that were retained. For example, the classification of application interface abstraction level was sufficient to predict many properties of that interface, such as the nature of data types exchanged across it. Hence, those properties did not need to be represented by separate dimensions.

At present, refinement through practical use seems the only way to remove such bugs from a design space. It may be that when more experience has been gained with software design spaces, patterns will emerge that will lead to a more theoretical, rigorous way of creating spaces. A thought-provoking observation about the present space is that the "top ten" functional dimensions just alluded to show a strong bias towards measures of system flexibility. Of the example dimensions in Section 2.2 (all drawn from the top ten), user customizability, user interface adaptability across devices, and application portability across user interface styles each measure a different aspect of flexibility. Requirements of this kind turn out to substantially outweigh any specific system properties (such as the nature or speed of I/O devices). Perhaps this is an artifact of the methodology, or a unique characteristic of user interfaces; but perhaps flexibility will some day be recognized as a fundamental determinant of many kinds of software structures.

7. Summary

This work attacks the problem of organizing software design knowledge to create routine design methods. Advances in this area promise not only to improve the basic process of software design, but to simplify a key task of software maintenance (namely understanding another person's design) and to provide a way of organizing the training of software engineers.

The underlying model of the design process is that one works from system requirements towards a completed design in several steps, or levels of abstraction. The particular design step considered here is the transition from high-level functional specifications to a gross system organization or architecture. We create a design space that describes the key functional specifications and the key structural choices to be made. Within this space, we formulate design rules that capture practical or theoretical knowledge about suitable choices for given requirements.

Since we view this technique as an engineering aid, the design rules need not be perfect to be useful. An informal set of rules (perhaps better called guidelines) can be useful simply by helping the designer to reject inferior choices quickly; this leaves more time available to consider and choose among the reasonable alternatives. In fact, even without rules the design space can be useful: it serves as a compact summary of different design approaches, and so can help the designer to avoid overlooking a good solution. Similarly, the functional side of the space reminds the designer of crucial considerations.

In the longer run, a reliable set of such design rules could serve as the basis for an automatic design assistant, or even for fully automatic system construction. The rule set experimented with here is a long way from that point, but seems already usable as informal guidelines of the kind just described.

The present work has investigated only one domain of software design, namely the architecture of user interface systems. This was an essential limitation in order to create a design space of manageable size. The author hopes that other researchers will undertake the task of building design spaces and rules for other domains---both other kinds of systems, and other levels of abstraction in the design process. Aside from being useful in their own right, such studies will show whether or not this approach really is a general-purpose method for organizing software engineering knowledge. Eventually, the combination of such studies may lead to the discovery of general principles about software design; for example, some kinds of design dimensions may be found to be universal.

Acknowledgements

Many of these ideas emerged from numerous discussions with Mary Shaw.

References

[Adobe 85] *PostScript Language Reference Manual*
Adobe Systems, Inc., 1985.
Published by Addison-Wesley.

[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.

[Landis 77] J. Richard Landis and Gary G. Koch.
The Measurement of Observer Agreement for Categorical Data.
Biometrics 33:159-174, March, 1977.

[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.
A Design Space and Design Rules for User Interface Software Architecture.
Technical Report CMU-CS-90-176, Carnegie Mellon University School of Computer Science, October, 1990.
Also available as CMU Software Engineering Institute Technical Report CMU/SEI-90-TR-22.

[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.

[Prieto 87] Ruben Prieto-Diaz and Peter Freeman.
Classifying Software for Reusability.
IEEE Software 4(1):6-16, January, 1987.

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

[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.

[Sherwood 88] Bruce Arne Sherwood and Judith N. Sherwood.
The cT Language and Its Uses: A Modern Programming Tool.
In *Proceedings of the Conference on Computers in Physics Instruction*. North Carolina State University, August, 1988.
Also Carnegie Mellon University CDEC Technical Report 88-28.

[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.