TOWARDS A PATTERN LANGUAGE FOR USER–INTERFACE DESIGN

by

RUNE ARNT SKARBO

A THESIS

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Master of Science

June 1990

APPROVED: [illegible]
Dr. Sarah A. Douglas

Abstract of the Thesis of

Rune Arnt Skarbo for the degree of Master of Science

in the Department of Computer and Information Science to be taken June 1990

Title: TOWARDS A PATTERN LANGUAGE FOR USER–INTERFACE DESIGN

Approved: [illegible]
Dr. Sarah A. Douglas

No effective methodology exists to assist user–interface designers in the process of translating high–level system requirements and functional semantics into user–interface design specifications. Traditional user–interface guidelines are too limited; they are too vague, do not specify how they can be realized, are not context sensitive, and insufficiently address program functionality. This thesis analyzes the feasibility of a user–interface design methodology beneficial to designers in the translation process. A user interface may be viewed as a patterned series––with the patterns describing sets of conflicting forces which occur in particular contexts, followed by specific configurations which stabilize the conflicting forces. Patterns can be used to map requirements and functional semantics onto user–interface design specifications. The bases for the 30 defined patterns are described. An example demonstrates how patterns are useful to translate requirements and functional semantics into user–interface design specifications.

VITA

NAME OF AUTHOR: Rune Arnt Skarbo

PLACE OF BIRTH: Lovelock, Nevada

DATE OF BIRTH: May 18, 1964

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon

DEGREES AWARDED:

Master of Science, 1990, University of Oregon
Bachelor of Science, 1988, University of Oregon

AREAS OF SPECIAL INTEREST:

Artificial Intelligence
User Interfaces

PROFESSIONAL EXPERIENCE:

Teaching Assistant, Department of Computer and Information Science, University of Oregon, Eugene, 1989–90

Programmer, IntelliCorp Corporation, Mountain View, California, 1989

ACKNOWLEDGMENTS

I would like to thank Professor Sarah Douglas, my adviser, for her encouragement and guidance throughout this work. Without her, this thesis could not have been written. To Professor Howard Davis, for his insightful comments on patterns and pattern languages, thanks are extended for the help and support. Thanks are also due to Professor G. Z. Brown for his comments on an earlier draft of this work. I am also indebted to the faculty, staff, and colleagues in the Department of Computer and Information Science for having contributed to making my stay here at the University of Oregon an educative and pleasant experience. Special thanks to Terry Chandler and Ray Vukcevich. Finally, I wish to express my sincere gratitude to my parents for their support, and to Anita, my wife, for her support, love, and patience.

DEDICATION

To Anita.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Designing computer applications is an extremely difficult task. Traditionally, the main difficulty was to achieve error–free algorithms. Software companies devoted considerable effort and expense in order to produce products that were close to error–free. Many companies had separate quality–assurance teams, or test teams, which often utilized state–of–the–art testing methodology (Chapman, 1981). And, prior to product completion, companies typically knew who the users of their products were going to be: people with a technical background who had the time and money to learn how to use the products.

Most of this is true today, as well. The present tense, instead of the past tense, could have been used in the previous paragraph. Producing error–free software is still very difficult, and software companies continue to invest a considerable amount of time and money in quality assurance. Today, however, the user group is typically not limited to people in engineering programs or technical institutions. People from a vast array of disciplines use computers. No longer are software producers able to assume that users are willing or able to spend days, or even months, to learn how to use a system. People want to use computers and computer programs as easily as any other tool encountered in daily life. Food processors, photocopy machines, telephone–answering machines, and typewriters are just a few examples of other such tools. One reason all these tools have become so popular is that their operation is fairly easy to learn. Consequently, if a computer system is to be perceived as a useful tool, then the system must be, at the very least, easy to use. Those who consider themselves computer experts have many times heard people say something to the effect that "how can you say that word processors are so fantastic? I'll stick to my typewriter--at least it does not take years to learn how to use it."

Today software companies have a new problem to consider: Programs have to be easy to learn and easy to use--the human/computer interaction must work smoothly. The system must be *user friendly*; that is, it must have a good *user interface*. Achieving effective user interfaces is just as difficult, if not more difficult, than achieving error-free code. Human/computer interaction has been primarily carried out using character-based displays with very limited graphical-display capabilities. Improved hardware technology has resulted in a shift from character-based displays to bit-mapped displays which have the ability to display detailed graphical images. There are also new means for communicating with the computer. For example, instead of relying only on the keyboard, it is now common for users to enter data using a pointing device called a *mouse*. The numerous additional elements increase the complexity of designing human/computer interfaces.

Designing Human/Computer Interfaces

The process of designing a user interface can be divided into five steps: (a) requirements analysis, (b) functionality analysis, (c) specification,

(d) implementation, and (e) evaluation. These five steps are ideally repeated until the user–interface design satisfies a predetermined set of standards or performance measures. The requirements analysis identifies the type of tasks that a system should be able to perform. It constitutes a high–level description of what the users wish to do with the system. Consequently, the users should be part of the development cycle from the very beginning. The functionality analysis results in a specification of the operations and the operational semantics. During the specification step, the presentation and the behavior of the user interface is determined. For example, decisions about menus, palettes, buttons, and so on are made. The fourth step involves the actual coding of the decisions made in the third step; and, in the fifth step, the interface is tested on future users.

Translating Requirements and Functionality into a Design Specification

One of the primary reasons behind why user–interface design is difficult is that there does not exist a methodology that effectively maps the system requirements and the functionality of a system onto a design specification. For the purposes of this thesis, the word *methodology* is meant to refer to a set of methods and postulates that indicate how some elements can be combined to form a whole. (This abstract description of a methodology will become more concrete later.) The translation process from functions to a design specification typically occurs under a set of constraints. For example, is the aim to please novice users, expert users, or both? And, what capabilities are provided by the system's hardware?

Moran (1981b) developed a command–language grammar that was intended to aid the user–interface designer in representing high–level goals, or requirements, and then transforming these goals into a specification. The command–language grammar is a design methodology in that it provides a process by which the system requirements can be transformed into descriptions that are at, or close to, the specification level. However, the problem is that the current format of the command–language grammar does not produce one specification, but rather an infinite number of specifications.

Jacob (1985) developed a user–interface specification tool termed *augmented transition networks*. These networks are useful for representing the transitions that occur in user interfaces at the specification level. However, Jacobs's networks are useful only after the requirements and functionality analyses have been carried out. A methodology that translates systems requirements and functionality into augmented transition networks does not exist.

Rapid prototyping tools have become popular as means for quickly producing and testing user interfaces. These tools enable the designer to move quickly from the requirements analysis to the interface evaluation. The time required for designing a user interface, therefore, may be shortened. A rapid prototyping tool does not constitute a user–interface design methodology, however. Methods for combining elements are not provided; and rapid prototyping tools neither assist in the identification of the evaluation criteria nor in the basic elements of a user interface.

The preceding discussion indicates that a methodology for translating a functionality description into a design specification is nonexistent. This thesis present an attempt to develop such a methodology. The methodology may at first remind the reader of traditional user–interface guidelines. However, as is argued in the next section, user–interface guidelines do not constitute a design methodology.

Problem Basis

The needs and the behaviors of computer users can be captured during the system requirements and functionality analyses. These are the first two steps in the user–interface design process. However, the needs and behaviors of the users are often lost in the process of translating the high–level functionality descriptions into a lower–level user–interface design specification. Since no known methodology for this translation exists, the consequence is that the needs and behaviors of the users are not sufficiently represented in the user–interface design specification.

The notion of capturing user needs and behaviors in a user–interface design is not new. For example, the following design principle is stated by Apple Computer Inc. (ACI): "*Users want to feel that they are in charge of the computer's activities*" (ACI, 1987, p. 4; emphasis in original). Obviously, if there is a need to be in control (over the computer's activities), then somehow this need should be reflected by the choices made in the design of the computer's user interfaces. ACI accommodates the user by encouraging a direct–manipulation interface style.

Another observation made by ACI (1987) relates to a different fundamental human trait: "*Users feel comfortable in a computer environment* *that remains understandable and familiar rather than changing randomly*" (p. 8; emphasis in original). In this case, ACI accommodates the user by providing a set of consistent user–interface elements, as well as a finite set of actions that can be performed on these elements.

Unfortunately, ACI's (1987) guidelines are not very useful for translating system requirements and functionality into a design specification. There are five primary reasons for this situation.

First, user–interface design guidelines provide little indication regarding the process of designing an interface. Where should the designer start? Should the process be top–down, bottom–up, or some combination of these processes? How can all the design principles be molded into a user–interface specification given the requirements and functional semantics? In other words, user–interface guidelines do not constitute a design methodology; and methods for combining the different principles are not provided.

Second, user–interface guidelines are typically expressed as a set of unrelated rules of thumb even though the elements of a user interface are, in fact, all highly interdependent. Thus, rules need to be made context sensitive. By establishing context–sensitive rules, the number of exceptions will decrease; and, the exceptions themselves will become context–sensitive rules.

Third, the Macintosh user–interface guidelines (ACI, 1987) restrict the elements of the user interface to typical Macintosh objects such as menu bars, buttons, scroll bars, icons, and so on. Only to a limited degree do the guidelines consider functions and features of a program as elements of an interface. No matter how successful a user interface is with respect to the performance measures outlined in Table 1 (which were derived from Moran, 1981a), the program as a whole will be unsuccessful if some basic functions and features are unavailable to the user. For example, people have a need to view data from different perspectives, or at different levels of abstraction; and some information is simply better understood at certain levels. The Page View and Print Preview options in Microsoft Word are examples of this. Page View allows the user to view a document in greater detail, relative to the regular on–screen view; the document is displayed in the exact format in which it will appear on the printed page. The Page Preview option, on the other hand, provides the user with a high–level view of the document; the size of the document page is reduced to fit the screen.

Fourth, most user–interface design guidelines are too vague. Even though the intention behind a certain guideline may be admirable, it often does not specify methods for satisfying the requirements. Consider, for example, the following design principles: "*Use concrete metaphors and make them plain, so that users have a set of expectations to apply to computer environments. . . . Communicate with the user in concise and simple terms*" (ACI, 1987, pp. 3, 14; emphasis in original). These guidelines will not be useful until questions such as the following can be answered: What exactly is a "concrete" metaphor? How far should a metaphor be taken? What exactly is a "concise and simple" term? (That which may be simple and concise to one user may be utterly confusing to another user.) A study by Furnas, Landauer, Gomez, and Dumais (1987) indicated that, if two people were to apply a term to a certain object, then the probability that they would relate the same term is less than .20.

Finally, user–interface guidelines do not indicate how tradeoffs and exceptions should be handled. The point is that the guidelines may appear to be very useful and clear on the surface, but attempts to satisfy the guidelines often result in hard–to–solve problems. An example of such a problem has been mentioned previously: How is it possible to make an interface both easy to learn (for novices) and efficient (for experts). Designing a user interface without deviating from guidelines is in practice very difficult.

TABLE 1. Some User–Interface Performance Measures

Categories	Performance Measures
Basic Performance Measures	Functionality: What functions are available to the user?
	Learning: Does user performance improve over time?
	Time: How long does it take to do a task?
	Error: Which types of errors are made, and how frequent are they?
	Quality: How good is the output?
	Robustness: How does the system adapt to unexpected conditions or new tasks.
Subjective Performance Measures	Acceptability: How does the user subjectively rate the system?
	Enjoyableness: How much fun is the system to use?
Extreme Conditions	Fatigue: How does performance degrade over time?
	Stress: How does performance degrade over adverse conditions?

This thesis primarily address the first three problems. A design methodology describing a top–down process that combines a set of interrelated elements in order to form a user–interface specification is explored. These elements are termed *patterns*. (A more in–depth discussion of this term is provided in Chapter II.)

Problem Statement

Currently, there exists no methodology that assists the designer in the process of translating system requirements and functional semantics into a user–interface design specification. (*Functional semantics* refers to the task that a function is supposed to accomplish, its preconditions and postconditions, and the objects on which the function operates.) There is a need to explore new ways to handle this phase of the user–interface design life cycle. User–interface guidelines have been and still are being used in this process. As indicated in the previous section, guidelines are of little use. In particular, a specific process for choosing and combining the guidelines has not been developed, methods for satisfying the guidelines have not been included, relationships among guidelines have remained unspecified, and system features typically are not addressed.

Proposed User–Interface Design–Specification Methodology

This thesis proposes the utilization of a user–interface design–specification methodology. The purpose of this methodology is to map user requirements, user needs, user limitations, and functional semantics onto a specification that covers aspects of the layout, the facade, and the functional syntax of a user interface. (*Functional syntax* refers to how a function should be presented to the user; for example, perhaps the function should be available from a palette rather than from a menu.) This methodology is based on the apparent similarities and closeness between user–interface design and architectural design. The proposed methodology is an adaptation of a methodology developed within the domain of architectural design. The main idea is that a user–interface design is composed of several patterns. Alexander, Ishikawa, and Silverstein (1977) have defined 253 patterns that can describe a virtually infinite number of architectural designs ranging in size from entire cities to the individual rooms of a house. A *pattern* is an instruction consisting of three basic elements: a context, a set of conflicting forces, and a configuration (see Figure 1). In other words, a pattern consists of a situation in which some problem occurs followed by a solution to the problem. This solution is given in terms of a certain attribute or feature which should be included in the design. Collectively, all the patterns form a language––a pattern language––that can be used to describe a particular design. (A further discussion of these three elements of a pattern is contained in Chapter II.)

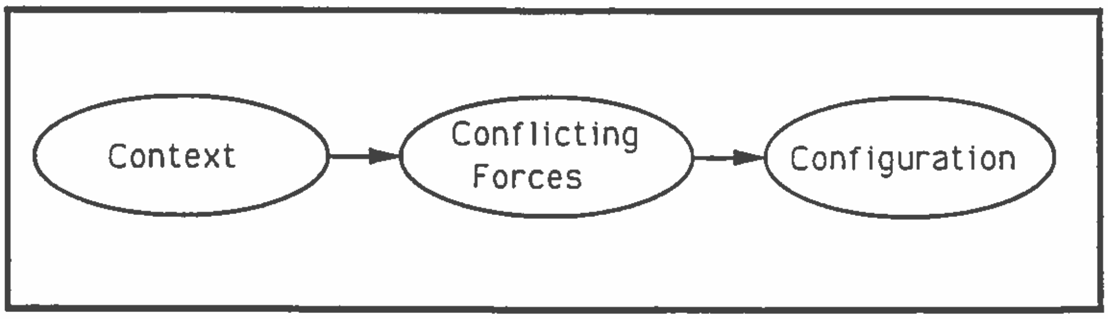


FIGURE 1. The Elements of a Pattern

It should be pointed out that Alexander, Silverstein, Angel, Ishikawa, and Abrams's (1975) language methodology is more than an intellectual exercise in design. The language has been used for an extensive number of small and large real–world projects. As a case in point, the University of Oregon campus and its buildings were designed using the pattern–language approach.

As was previously presented, there is a problem in that a set of user–interface guidelines does not constitute a design methodology in any manner. However, a set of patterns, together with a process that specifies how to use and combine them, does constitute a design methodology. The patterns in a pattern language are interdependent; and there are patterns of varying sizes. Chapter III contains a demonstration of how the patterns actually form a network. That is, the largest patterns depend on some smaller patterns, and the smaller patterns depend on even smaller patterns, and so on. For each pattern, the relevant interdependencies are specified. Patterns typically address user–interface elements such as windows, palettes, menus, the layout of the user interface, and features that should be included. The patterns and the processes for combining the patterns so that they form a user–interface design are presented within this thesis. The mapping from system requirements to a design specification is made possible by stating the requirements in terms of problems that can be addressed by the patterns.

Another problem, also identified earlier, concerns the fact that user–interface guidelines are too vague. A pattern language for user–interface design is more specific in that the patterns provide solutions to specific problems. Included in a pattern is a precise specification of the context in which it is relevant. Furthermore, a specific problem that occurs in that context is described; and, finally, a specific solution is given.

It has been additionally argued that user–interface guidelines do not sufficiently address the functionality of a system. The domain of guidelines is typically limited to the graphical elements of a system (e.g., the windows, menus, icons, etc., on the Macintosh). Patterns, on the other hand, are based on human needs and human limitations. These needs and limitations can be addressed by selecting particular interface layouts and elements, and by incorporating certain functions and features into a system.

The most interesting aspect of the pattern–language approach is that it will eventually constitute a design methodology. In most fields, it is often the case that people are equipped with all the tools necessary to accomplish some goal; but, because these tools frequently exist as fragmented and often unrelated pieces of knowledge, principles, or whatever, the task becomes how to select the needed tools and appropriately string them together. This problem also exists in the field of user–interface design. A designer may comprehend a large set of design principles, but fail in the process of combining these principles into a coherent user–interface design. By using the pattern–language approach, this problem is reduced.

Solution Constraints

The methodology proposed herein captures only a few aspects of the translation process from system requirements to a design specification. This thesis is, first and foremost, a feasibility analysis of the pattern–language approach as opposed an effort to present a complete functioning system. The goal was to determine whether or not the pattern–language approach is worth studying and refining in order to achieve a more complete method for user–interface design specifications. (Chapter II emphasizes that a pattern language never can be complete. That is, patterns change as human beings change; and, as more knowledge is gathered in regard to human needs and behaviors, patterns are added, deleted, and modified. Thus, the development of patterns is a perpetual process.)

The bases for the patterns presented in this thesis often lack support by empirical studies. This is not because empirical studies have indicated different results; but, rather, it is due to the fact that research has yet to be carried out in many of the relevant areas. Ideally, a pattern should address all the conflicting forces that are present in a given situation, and the validity of the solutions should be justified by empirical studies. A related obstacle is that there is no reliable way to determine exactly which forces are present in a given situation. This does not mean that it is impossible to discover efficient patterns; in fact, the proof that this is possible is provided by Alexander et al. (1977). The patterns in this thesis have been based on several sources. The conflicting forces have been identified by using the following methods:

1. By observing novice users and identifying the difficulties they experience, a common set of problems emerged.

2. By relating already–identified problems in architectural design to user–interface design, important elements have been revealed.

3. By personal experience and by conversing with a user–interface design expert, essential theories were generated. In addition, to obtain a

more complete understanding of the pattern–language approach, interviews were conducted with an architect who uses pattern languages. (The principal findings from these interviews, which were recorded on videotape, are described in Appendix A.)
Some of the solutions to the conflicting forces were obtained from existing literature. Other solutions were based on personal experience.

Again, this is a feasibility analysis. The author does not claim to have identified infallible patterns. Some patterns may capture all the conflicting forces and suggest a good solution; others may not. Alexander et al. (1977) devoted more than 12 years to identify their 253 general patterns for architectural design. However, the important point is that, given a set of patterns that are based on an underlying theory (and a method that specifies how to use these patterns), the end result is a user–interface design–specification methodology. The evolution of patterns is a process that must be subsequent to this feasibility analysis.

Summary

This introductory chapter motivates the need for new and improved ways to assist the process of translating system requirements and functional semantics into a user–interface design specification. A methodology based on Alexander et al.'s (1977) pattern–language technique has been proposed as a process for eliminating some of the deficiencies which make user–interface guidelines incapable of generating a design specification given a set of high–level descriptions.

Chapter II presents an introduction to the theory behind patterns and pattern languages. Chapter III explicates the existence of and foundations for user–interface patterns, and specifies the methods for identifying patterns. In Chapter IV, a small number of user–interface patterns are identified. Chapter V presents an example that demonstrates how system requirements can be translated into a set of patterns that represents a user–interface design specification. Chapter VI includes a discussion of the feasibility of the proposed methodology and some issues which are important for future research. Of particular interest is the idea of incorporating the design methodology into an expert system in order to automate the translation of system requirements into a user–interface design specification.

CHAPTER II

PATTERN LANGUAGES FOR ARCHITECTURAL DESIGN: AN OVERVIEW

In Chapter I, it was argued that there is a need for a methodology that can translate system requirements and functional semantics into a user–interface design specification. Deficiencies of typical user–interface guidelines were discussed; specifically, they are too vague, they are not context sensitive, and they do not address the functionality of a system. A new design methodology that can translate system requirements and functional semantics into a user–interface design was proposed. This new design methodology is based on Alexander et al.'s (1977) work in the field of architectural design. In this chapter, the work by Alexander (1979) and Alexander et al. (1977) is reviewed in order to establish a foundation for the adaptation of this methodology to user–interface design.

Buildings That Work

Most people have been in cities, inside buildings, or in places that have some qualities which generate a feeling of aliveness––in short, these areas make people feel good. Alexander's (1979) thesis is that there is a timeless way of building that is so fundamental, so close to the needs of human beings, that it would be impossible to create great towns, buildings, or houses without following this way of building. Yet, this does not imply that there is only one way to proceed in terms of constructing physical structures. It simply means that, among the infinite number of variations that exist to design buildings, there is some invariant core common to them all. There are some elements that need to be present in a building in order for that building to work. Furthermore, there is a sequence of events that forms the basis for the act of building. The elements of this sequence are definable, as is the context in which the elements apply. To specify these elements, the level of analysis that is required hinges upon a representational form which reveals that all possible processes of construction are versions of a deeper process.

First, there must be a way to examine the ultimate building blocks of the environment. Each town, each building, and each room is comprised of these building blocks. Alexander (1979) has shown that towns, buildings, and houses are constructed from certain entities called *patterns*. Understanding such structures in terms of patterns provides a way to identify what it is that makes the buildings in a town similar. In the following section, Buildings as Patterns, the feasibility of such an identification is discussed. Viewing the environment in terms of patterns cultivates the capability to understand exactly that which makes a building alive, and that which makes a building dead.

Second, the processes that generate patterns must be understood. This involves recognizing how patterns come about, and is discussed in the next section which outlines Alexander's (1979) theory that a pattern is the result of certain combinatory processes.

Buildings as Patterns

In order to define the elements (or patterns) of towns and buildings, Alexander (1979) argues that every place is provided with its character by the patterns of events that happen within it. The shape of a town or the shape of the building are irrelevant; rather, it is the episodes which occur therein that are of significance. In other words, function precedes structure. However, it is impossible to think about certain events without also thinking about where the events are happening. For example, it is impossible to imagine doing dishes without also imagining doing dishes *somewhere*. This *somewhere* could be many places, but all the places would have a number of common attributes. For instance, soap and water would be available, and there would be separate places to store washed and unwashed dishes. It is also natural to think about a place in conjunction with the episodes that happen there. For example, it is impossible to think about a kitchen without thinking about preparing food, eating, sitting around the kitchen table, chatting, and so on. Function and structure interact.

It is given that towns and buildings acquire their characters from patterns of events, and that somehow these events are correlated with space; but, then, how exactly are events and space linked together? Knowledge about how the structure of space supports the episodes that happen there is vital. This knowledge should be strong enough so that the changes in the patterns of events could be predicted if the structure of the space was changed. This would constitute a theory that addresses the interaction between space and events.

A Pattern is a Relationship Between Other Patterns

At first, it might seem as if the building blocks of a town include houses, gardens, shops, workplaces, and parking lots. Similarly, it might seem as if the essential building blocks of a house are walls, windows, rooms, ceilings, stairs, door handles, and so on. However, breaking space up into these elements does not explain how or why the elements are associated with patterns of events such as cars and buses driving in the street, families living in houses, and people walking through doors. Moreover, if those were the building blocks, why are they different every time they occur? Every town is different from all other towns, every house is different from all other houses, and every room is different from all other rooms. Because the elements differ every time they occur, they cannot be the ultimate building blocks of space. There must be some other invariants throughout the endless variations that constitute the atomic building blocks. Instead, the structure of the space that a town or a building consists of must be examined. What exactly is it that is repeated?

There are relationships between the elements that keep repeating. Alexander (1979) uses the following example:

> Consider a typical mid–twentieth–century American metropolitan region. Somewhere *towards the center* of the region, there is a central business district, which contains a *very high density* office block; near these are *high density* apartments. The overall density of the region *slopes off with distance from the center, according to an exponential law; periodically there are again peaks* of higher density, but smaller than the central ones; and *subsidiary* to these *smaller* peaks, there are still smaller peaks. Each of these peaks of density *contains* stores and offices *surrounded by* higher density housing. (p. 86; emphases in original)

This excerpt illustrates that a structure, be it a town or a building, is comprised of patterns of relationships. These relationships are components of the elements in our environment. In fact, not only are the relationships linked to the elements, but also the elements themselves are patterns of relationships. The reasoning is that, once it is understood that an element is part of the pattern of relationships between the element and the things around it, a greater realization can be achieved. The element is not merely embedded in a pattern of relationships, but is itself a pattern of relationships. Hence, the elements that were considered first can be viewed as labels for the patterns that actually do repeat––namely, the patterns of relationships. Alexander (1979) considers each of these patterns as a morphological law. In this domain, a morphological law defines a set of relationships in space. In other words, given some context X, the parts A, B, . . . are related by the relationship r, $X \rightarrow r(A, B, \ldots)$. Alexander provides two examples.

> Within a Gothic cathedral → the nave is flanked on both sides by parallel aisles. . . .
> Where a freeway meets an artery → the access ramps of the interchange take the rough form of a cloverleaf. (1979, p. 90)

Each of these laws are patterns of relationships made up of other laws. Examining a pattern reveals that it does not merely consist of parts, but that these parts are patterns as well. Alexander uses the following example to illustrate this point:

> Consider, for example, the pattern we call a door. This pattern is a relationship among the frame, the hinges, and the door itself: and these parts in turn are made of smaller parts: the frame is made of uprights, a crosspiece, and cover mouldings over joints; the door is made of uprights, crosspieces and panels; the hinge is made of leaves and a pin. (1979, p. 91)

These things that first were considered parts are also patterns. Even though each part may take on an infinite number of shapes, the relationships (which make them patterns) are not lost.

The Connection Between Patterns of Space and Patterns of Events

Intuitively, each pattern in space has an associated pattern of events. Consider, for example, a pattern that describes a kitchen. The associated pattern of events includes the way in which the kitchen is used; for example, food is prepared, food is consumed, and dishes are done. It is also intuitively apparent that a pattern of space does not cause a pattern of events; and, a pattern of events does not cause a pattern of space. (The consideration of space and events together as a pattern leads to the realization that a pattern is basically invented by people's culture.) However, the pattern of space is what allows the patterns of events to happen; and the pattern of space maintains the patterns of events. Since the character of a town or a building develops from the pattern of events, and the pattern of space is a precondition for the pattern of events to occur, the pattern of space must necessarily be one of the requirements that provides a town or a building with its character.

What is most remarkable is that a town or a building consists of relatively few patterns. Alexander (1979) claims that a town (such as London, England) can, in essence, be defined by a few hundred patterns. Patterns can be combined in a practically infinite number of ways. Although the patterns can be combined in a numerous ways, the underlying invariants are still present.

Using a System of Patterns

Consider the question "what is it that an individual farmer did, when he decided to build a barn, that made his barn a member of this family of barns, similar to hundreds of other barns, yet nevertheless unique" (Alexander, 1979, p. 176). First, it may be surmised that the farmer paid attention only to the function of the barn, and so the barn is beautiful as a result of the farmer being in touch with its function. But that does not explain why a specific farmer's barn is similar to other barns. That is, if all barns were created on the basis of their function, a much greater variety in design would be expected. Perhaps the farmer is copying the other barns that he or she knows about. But this does not explain why there is such a great variety of barns. The answer to this question, according to Alexander (1979), is that the farmer is able to build a barn because every barn is comprised of patterns. Hence, the farmer *is*, in fact, copying other barns. But the farmer is not copying complete drawings of other barns; the farmer is copying a system of patterns.

How Patterns Work

A pattern describes what to do in order to generate the physical structure it defines. But there is more. A pattern exists to solve some problem. It is not merely a pattern that might or might not be used; a pattern is a description of something that *should* be used in some particular context. The fact that a pattern is context sensitive is extremely important. Given a specific context, certain problems arise. The pattern describes the context and the problems; and, in addition, the pattern provides a solution to the problems. Several patterns, or a system of patterns, can be said to form a language. A *language* may be defined as a set of elements and a set of rules for combining the elements. In a pattern language, the patterns constitute the elements. Each pattern describes how it itself is a pattern comprised of still smaller patterns. In addition, there are rules embedded in the patterns which describe how they can be realized, as well as how they can be arranged and combined with other patterns. It should be noted that the rules and the elements are indistinguishable; the rules are part of the patterns. Yet, there is a close relationship between ordinary language (natural language) and a pattern language. Both ordinary languages and pattern languages can be used to form an infinite number of variations, with each variation being appropriate to the different contexts in which they are formed (see Figure 2).

Natural Language		Pattern Language
Words	⟷	Patterns
Rules of grammar and meaning	⟷	Patterns that specify relations among patterns
Sentences	⟷	Places and buildings

FIGURE 2. Natural Languages Versus Pattern Languages

Discovering Patterns

In order to create a pattern language, the ability to "learn how to discover patterns which are deep, and capable of generating life" is crucial (Alexander, 1979, p. 243). It should be recalled, from the previous discussion of the elements of a single pattern, that a pattern is a rule with three parts. It expresses a relation between a context, a problem that occurs in that context, and a solution to the problem. A pattern can deal with nearly any type of problem. The following list of problems will provide some intuition as to what types of problems are addressed in a pattern:

> ENTRANCE TRANSITION resolves a conflict among inner psychic forces.
> MOSAIC OF SUBCULTURES resolves a conflict among social and psychological forces.
> WEB OF SHOPPING resolves a conflict among economic forces.
> EFFICIENT STRUCTURE resolves a conflict among structural forces.
> GARDEN GROWING WILD resolves the conflict between forces of nature, the natural growing process in plants, and people's natural actions in a garden. (Alexander, 1979, p. 248)

How, then, can a pattern be *identified* or *created*? (Notice that a distinction is made between identifying and creating patterns.) First, when identifying a pattern, it has been already established that there is something about a place that is worth abstracting. For example, it is worthwhile to spend some time and energy on identifying why a certain place makes people feel good. That is, one wishes to determine what the qualities are that make that place work. When creating a pattern, the goal is to define some physical feature that does not yet exist. After having identified a set of conflicting forces, the objective is to create some physical features that resolve the conflicts. The processes of identifying and creating patterns are thus somewhat different.

Second, when identifying a pattern, the problem must be defined. In other words, after identifying some physical feature in the first step of the process, now the conflicting forces which the pattern resolves must be identified. When creating a pattern, some physical properties that will solve the problems identified in the first step must be identified.

Finally, in both cases, the range of contexts in which the pattern holds must be defined. Included in this is a specification of the contexts in which the system of forces exists. In addition, it must be determined whether the new pattern has any side effects. If it has created any new problems, patterns that solve these problems must be identified or created.

As will become more apparent in Chapter III, pattern creation (rather than pattern identification) is typically used in the process of developing user–interface patterns. The scheme identifies some conflicting forces, and then specifies how the conflicting forces may be resolved. Interestingly, a large portion of the solutions arrived at are embedded in many existing user interfaces.

Using the Language

In the previous sections of this chapter, the theory behind the existence of patterns was reviewed. Over a period of more than 12 years, Alexander et al. (1977) identified 253 general patterns. For each pattern, the following seven parameters were specified:

1. A picture. Each pattern is exemplified with a picture.

2. An introductory paragraph. This paragraph sets the context for both small and large patterns; this paragraph also specifies how the current pattern assists in the completion of larger patterns.

3. A headline. The headline is a succinct specification of a particular problem (or set of conflicting forces).

4. A body. In this division, the validity of the pattern is addressed. If there are different ways to use the patterns, they will be included here.

5. A solution. This is a succinct instruction on how to build the pattern.

6. A diagram. Each solution is demonstrated by a diagram.

7. A reference to smaller patterns. The last parameter ties the pattern together with other smaller patterns. This part specifies which smaller patterns need to be considered in order for the current pattern to be complete.

By associating these pieces of information with each and every pattern, the interconnectivity of the patterns becomes clear. Moreover, by stating the problem and its solution, a pattern can be modified to fit specific needs if that becomes necessary.

The patterns are ordered beginning with the largest ones. Large patterns include those for entire cities and communities. Next there are patterns for neighborhoods and clusters of buildings. Even smaller patterns entail buildings and rooms inside buildings. The smallest patterns specify the details of construction––such as roofs, floors, and room layouts. Every pattern depends not only on some particular set of larger patterns in the language, but also on certain smaller patterns. In other words, a pattern is supported by a set of larger patterns, and it is embellished by set of smaller patterns. The important point is that an entity cannot be built in isolation. The world around it and within it are equally important.

The ordering of the patterns is of great significance. During the design process, some patterns need to be realized or implemented before others. For example, the shape of a building needs to be specified before the room layouts are determined. Each pattern can be considered as being in the center of a network. Intuitively, in the design process, the larger patterns should be considered before the smaller patterns. This would imply a top–down design approach. (In Chapter VI, an explanation is provided to demonstrate that a strict top–down approach is unrealistic; an effective methodology should allow for a mixture of top–down and bottom–up designs.)

Alexander et al. (1977) suggest that a pattern should be treated as an hypothesis. A pattern is an attempt at detecting a solution to a particular problem. Each of Alexander et al.'s solutions were rated on a 3–point scale, and the rating was dependent on whether or not the authors thought they had succeeded in finding a true invariant. For example, the pattern SOUTH FACING OUTDOORS, has the highest possible rating (two stars). The problem statement is that "*people use open space if it is sunny, and do not use it if it isn't, in all but desert climates*" (Alexander et al., 1977, p. 514; emphasis in original). The solution is to "*always place buildings to the north of the outdoor spaces that go with them, and keep the outdoor spaces to the south. Never leave a deep band of shade between the building and the sunny part of the outdoors*" (Alexander et al., 1977, p. 516; emphasis in original). Few architects would disagree with this pattern.

On the other hand, the pattern HALF–PRIVATE OFFICE has the lowest possible rating (zero stars). The problem concerns "*what is the right balance between privacy and connection in office work?*" (Alexander et al., 1977, p. 717; emphasis in original). The solution is the following:

> *Avoid closed off, separate, or private offices. Make every workroom, whether it is for a group of two or three people or for one person, half–open to the other workgroups and the world immediately beyond it. At the front, just inside the door, make comfortable sitting space, with the actual workspace(s) away from the door, and further back.* (Alexander et al., 1977, p. 718; emphasis in original)

In this latter case, Alexander et al. were not contending that their solution constitutes a true invariant. In other words, they believed that there was room for exploration in finding a better solution to this problem.

It is interesting how difficult it is to solve problems that arise due to tradeoffs. Tradeoff problems are difficult to unravel in the architectural domain, as well as in the domain of user–interface design.

Alexander et al. (1977) describe a procedure for how the language should be used. The idea is to isolate the patterns that will be part of the language that describes a particular project. There are eight steps:

1. Compile all existing patterns into list. (This list is used in the second step.)

2. Select the pattern that best describes the overall scope of the project. For example, if the project involves building a porch onto the front of the house, the pattern PRIVATE TERRACE ON THE STREET is highly relevant.

3. Read the description of the pattern selected in the second step. Some larger patterns may be mentioned in the description; these are the patterns which are candidates for the language. Larger patterns obviously should not be

included if there is no way of enforcing them or no manner in which to create them. However, all the smaller patterns probably will be important. Place all the small patterns in the list of patterns that will constitute the language for the project.

4. At this point the list of patterns will have grown. Starting with the largest pattern in the list, read the instructions for that pattern, and (as in the third step) extract all the patterns that are relevant for that particular pattern.

5. If in doubt about a certain pattern, do not include it.

6. Repeat the fourth and fifth steps until all the patterns for the project for the project have been identified.

7. At this point, personal patterns may be added to the list. There may be patterns which are required, but are not in the list.

8. If a pattern needs to be changed, change it.

An Example

After a set of patterns has been chosen for a particular project using the eight steps specified in the previous section, the process of unfolding a design may begin. For example, assume that the goal was to create a garden. The eight–step process would produce a series of patterns, all of which are part of a network (see Figure 3). Unfolding a design implies *implementing* or *realizing* a design. Implementing a set of selected patterns is also a step–by–step process, wherein each step brings exactly one pattern to life. High–level patterns are realized before lower–level patterns. In the example in Figure 3, the HALF–HIDDEN GARDEN pattern would be first implemented. (Instructions on implementing a pattern are an integral part of the pattern itself; this is further explained in Chapter IV.)

Appendix A includes further information concerning the pattern–language design process and an example of how a pattern is developed on the basis of a set of requirements.

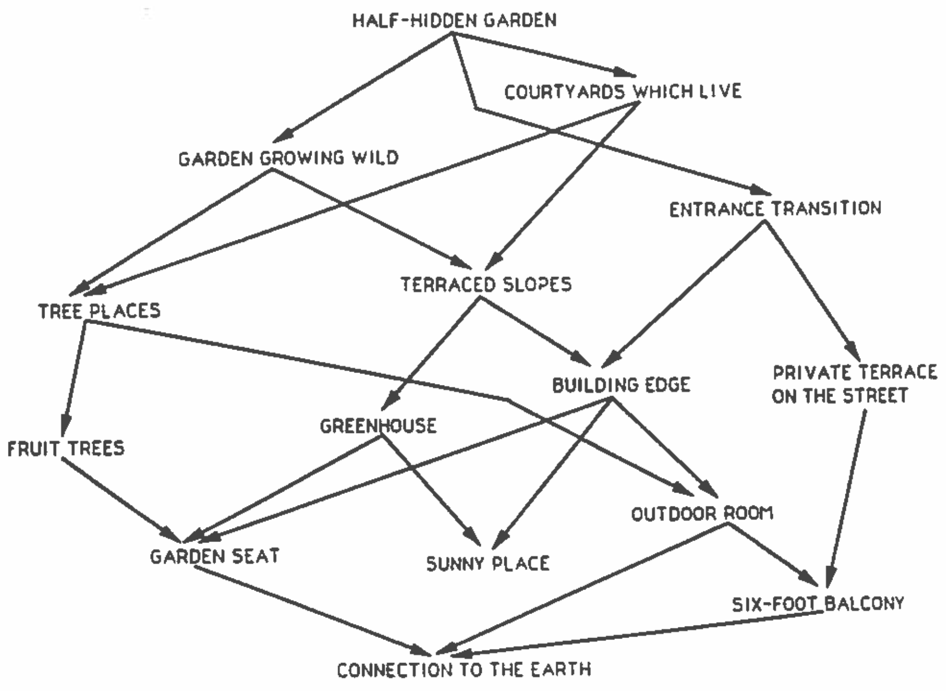


FIGURE 3. A Network of Patterns Representing a Garden of Patterns

The Users of the Language

The pattern language was developed with several intentions in mind. First, it was believed that it would assist nonarchitects in creating good designs; applying the language requires no prior knowledge about the architectural domain. Second, the language should allow people in neighborhoods and communities to design together. People would be able to communicate their ideas using a common language. It could also be used as a way for architects to communicate with clients. Finally, even though many of the principles in the language are second nature to most architects, the set of patterns can function as an encyclopedia for those who want to confirm their beliefs or refresh their memory about some of the less obvious principles.

Advantages and Disadvantages

The main advantage with a pattern language is that it provides the user with a set of patterns which enhances architectural design. It allows nonarchitects to design on their own, and to participate in the design of their communities. A pattern language can be used as a means for communication.

The main disadvantage is that the patterns are very general and, therefore, may not suffice for a particular project. In such a case, Alexander et al. (1977) propose that *personal* patterns should be created, and they argue that actually this freedom is what makes the language very powerful. But if the nonarchitect creates personal patterns, there obviously is no guarantee that the patterns are good patterns. It has been previously discussed that Alexander et al. (1977) spent nearly 12 years identifying 253 general patterns. If, however, a pattern is merely a slight variation of an already existing pattern, the task may be easier than creating a pattern from scratch. For example, a project may involve a sauna. Although there is no pattern in Alexander et al.'s base set of patterns that addresses the sauna component, there is a pattern called BATHING ROOM which could be used as a template to create a pattern called SAUNA.

Controversies

There is a significant amount of controversy surrounding the pattern–language design methodology (G. Z. Brown, personal communication, May 1990; H. Davis, personal communication, February 1990). Some architects disagree with many of the patterns, as well as with the idea that architectural design can be captured by a set of patterns. Moreover, identifying new patterns when no existing pattern suffices is not a trivial matter; and this problem reduces the usability of the language. Some architects also feel that their profession should not be in the hands of the layperson.

However, the language and its principles should be taken seriously. It can be argued that architects make glaring mistakes in their designs, and that these mistakes could have been avoided if the patterns had been consulted. What is meant by a *mistake* in architectural design, and who decides when a blunder has been made? Obviously, if a house falls down or a bridge does not hold up, a finger can be pointed at the architect. But this type of mistake is not relevant in this discussion. It is the mistakes that prevent a design from *working* that are of interest. Few architects would argue against the pattern SOUTH FACING OUTDOORS. Nevertheless, the Bank of America building in downtown San Francisco has its plaza on the north side. During lunchtime, the plaza is empty; there is no sun. People sit on the south side of the building where the sun is during the noontime, even though the south side of the building was not designated as a place for people to meet, socialize, and relax. The plaza does not work.

Summary

This chapter has established the primary foundation for this thesis. The details of the pattern–language design methodology (Alexander, 1979; Alexander et al., 1977) have been reviewed, and it has been shown that a pattern is a solution to a problem that occurs in some particular context. The process of how a particular design project can be matched against an existing catalog of patterns was described. The eight–step algorithm which was presented assumes that the user is able to identify at least one pattern that is central to the project at hand. Once this pattern has been defined, the inclusion of other patterns follows from the structure of the language. A difficulty arises when none of the existing patterns apply. In such cases, new patterns have to be identified or created. The differences between pattern identification and pattern creation was addressed; the relation between natural languages and pattern languages was demonstrated; and the connection between patterns of space and patterns of events was identified. Finally, it was established that the most troublesome disadvantage with patterns is that it is often very difficult to create good patterns.

CHAPTER III

A PATTERN LANGUAGE FOR USER–INTERFACE DESIGN

In Chapter II, the concept of a pattern language was introduced. The basis for such a language and the underlying philosophy were discussed. This chapter presents the foundations for a pattern language for human/computer interface design.

First, the foundations for the user–interface patterns (UIPs) that have been defined are discussed. (The specific patterns are separately listed in Chapter IV.) Some UIPs were identified in the same way that architectural patterns were identified. For these UIPs, it had been already established that there was something about a user interface worth abstracting. The task was to first identify the good qualities, then to discover the problems that these qualities solved; and, finally, the relevant contexts for the UIP had to be determined. Other UIPs were discovered using different approaches. For example, by observing users, a number of problems in the human/computer interaction became evident. After a problem had been specified, the conflicting forces that created the problem had to be identified. Then some properties that would eliminate the conflicting forces had to be found, followed by a specification of the relevant contexts.

Second, the elements that constitute a UIP are described. The UIPs are not completely analogous to architectural patterns; the most striking difference is that a UIP does not always have an illustration that represents an archetypical example of the UIP. (The reason for this difference is discussed in Chapter VI.)

Third, an argument is provided regarding why UIPs are not merely guidelines similar to those described by, for example, ACI (1987) and Smith and Mosier (1986); instead, a design–specification methodology is given.

Finally, the properties that make a collection of UIPs a language, or a design language, are described.

Foundations

Form Follows Function

Chapter II discussed the interaction between space and events, or form and function. This issue is relevant not only in the architectural domain, but also in the domain of user–interface design (Hooper, 1986). Just as many architects do not consider the facade of a building effective unless it reveals the building's function, a computer interface will not be effective unless the functionality of the system is revealed. People can walk into a building to evaluate the informativeness of the facade. This can be said about user interfaces as well; that is, the user can "walk into" the design. The problem is, however, that if the user takes a step in the wrong direction, it is often difficult to "back out of" the design. The user only has the information displayed on the screen. In user–interface design, careful consideration must be given to how the user is informed about a system. It is crucial that the basic components of the interface are chosen in a manner that gives the user a general idea about the functionality of the system.

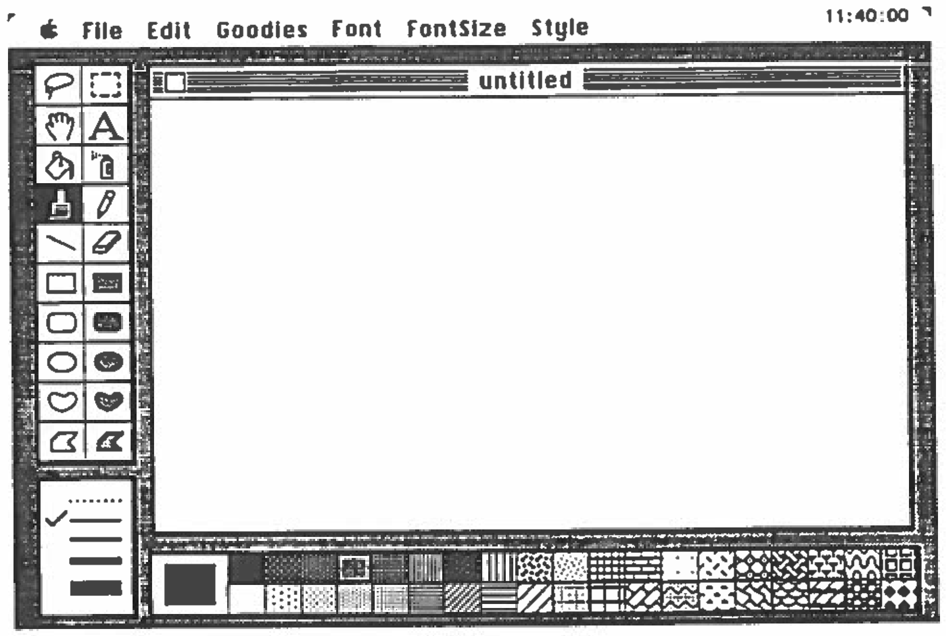
During the course of this study, elements as fundamental and seemingly trivial as a *palette*[1] was studied among other aspects. It was questioned how a palette might affect user behavior; whether a constantly displayed palette is *better* or *worse* than a palette placed in a pull–down menu; the types of predictions which can be made with respect to palettes; and whether a theory could be developed which would address the relationships between user behavior and palettes.

A consideration of the following simple example illustrates some of the issues involved. With respect to novice users, it is probably safe to say that constantly displayed palettes are better than palettes which are placed in pull–down menus. Palettes seem to encourage users to explore a system on their own because the functionality of a system is better revealed––it is right there on the screen. Consider the differences between the old and new versions of MacPaint (version 1.4 versus version 2.0). Even though the old version has many flaws, the interface reveals the basic functionality of the program (see Figure 4a). The interface of the new version, however, is ineffective in this respect (see Figure 4b).

To informally test the informativeness of the two MacPaint interfaces, 10 people who had never used a Macintosh before were gathered; 5 people used the old version, and 5 people used the new version. They were asked to draw a rectangle using MacPaint. Each person, in turn, was asked to explore the

[1]A palette is a collection of small symbols, usually enclosed in rectangles. A symbol can be an icon, a pattern, a character, or a drawing that stands for an operation. When the user has clicked onto one of the symbols, it is distinguished from the other symbols, and the previously selected symbol goes back to its normal state.

(a)



(b)

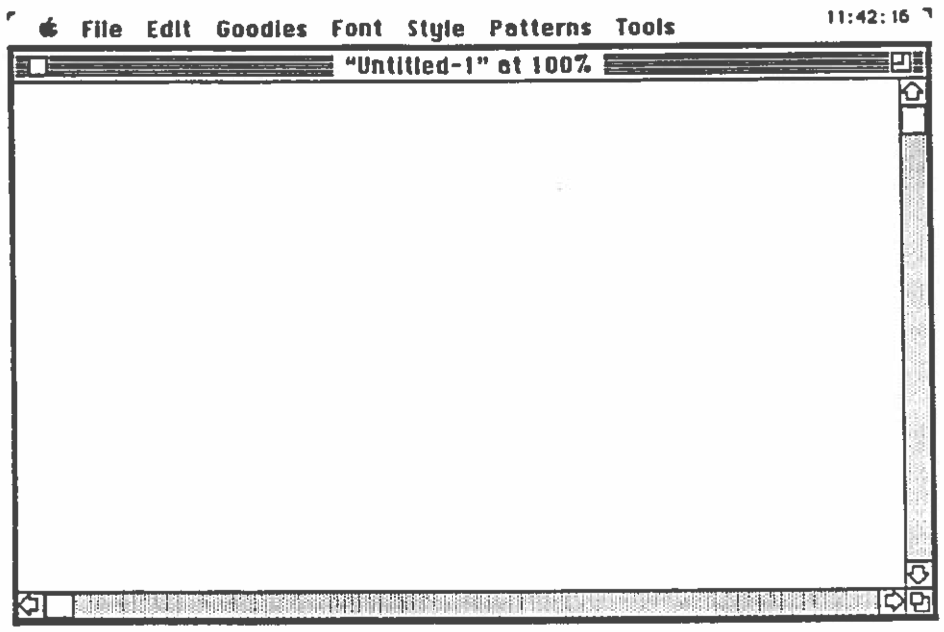


FIGURE 4. Old and New Versions of the MacPaint User Interface (*a* portrays the old version of MacPaint, *b* portrays the new version)

system until the goal had been accomplished. The subjects who worked with the old MacPaint version finished the task in an average of 55 seconds. The other (new–version) subjects finished the task on an average of 125 seconds.

By collecting large amounts of the type of information described in the preceding paragraph, a theory that relates the presentation of a user interface to the behavior of the user can be formed. In a pattern language for user–interface design, such issues are addressed. A pattern language will eventually constitute a theory regarding how the design of an interface affects user behavior. One of the ultimate goals of developing a pattern language is to provide the designer with a set of hypotheses, or UIPs, that should be included in an interface. These UIPs are fundamental with respect to the relationships among user psychology, user–interface elements, and system functionality.

Identifying User–Interface Elements

One prerequisite for developing UIPs is the ability to identify the elements that culminate in a successful user interface. That is, one must determine whether or not successful interfaces share any common elements. Just as barns were found to be similar but yet unique (as described in Chapter II), perhaps successful interfaces also have such a property. Although all interfaces are somewhat different, interesting similarities can be detected. These similarities include, for example, the manner in which a certain class of functions is presented, the method with which a certain class of functions is activated/ deactivated, how functionality satisfies the needs of the user, and how an interface adheres to the limitations of human cognition. The UIPs, herein, address the manner in which interfaces are able to resolve the conflicting forces that involve user psychology, user–interface elements, and functionality. The problem of identifying and creating UIPs is approached from two angles.

Human Psychology as a Basis for User–Interface Patterns (UIPs)

One of the most important foundations for Alexander's (1979) patterns is human psychology. In fact, many of his patterns are direct results of observations related to the needs and desires of human beings. In like manner, many of these needs are directly related to user–interface design as well. Two examples illustrate this comparability.

Alexander et al. (1977) definition for a pattern named YOUR OWN HOME is that "people will only be able to feel comfortable in their houses, if they can change their houses to suit themselves, add on whatever they need, rearrange the garden as they like it . . ." (p. 394). The desire to personalize one's home is related to personalizing a user interface. An appropriate example is the Macintosh desktop. Users are allowed to arrange the different elements on the screen in accordance with personal preferences. Imagine how frustrating it would be for users if they had been prohibited from moving icons, desktop accessories, and windows to the positions of their liking. (In fact, the ability to rearrange user–interface elements is so important that a UIP which addresses this need––entitled UIP 7: PERSONALIZABLE SCREEN LAYOUT––is defined in Chapter IV.)

Alexander et al. (1977) have defined a pattern labeled FAMILY OF ENTRANCES. The problem the pattern addresses is the following:

> *When a person arrives in a complex of offices or services or workshops, or in a group of related houses, there is a good chance that he will experience confusion unless the whole collection is laid out before him, so that he can see the entrance of the place where he is going.* (p. 500; emphasis in original)

And the solution may be explained in the following manner:

> *Lay out the entrances to form a family. This means:*
> 1. *They form a group, are visible together, and each is visible from all the others.*
> 2. *They are all broadly similar, for instance, all porches, or all gates in a wall, . . . are marked by a similar kind of doorway.* (Alexander et al., 1977, p. 502; emphasis in original)

This pattern can be related to palettes and pull–down menus. All menus are visible at all times, and it is often the case that the menus as well as their corresponding items can be accessed from most program states. However, more subtle analogies can be drawn from this pattern. Consider the program SuperPaint which is fairly complex with respect to functionality; it provides a large number of options to the user. In addition, the manner in which the program presents the large number of choices available in one palette is interesting. SuperPaint actually combines two palettes into one: one for drawing functions, and the other for painting functions (see Figure 5). Having two separate palettes or a single larger palette would have increased space usage and may have caused added confusion due to the larger number of choices displayed at the same time. The interface designers needed to convey to the user that the mode of the palette was adjustable; they accomplished this by using partly overlapping toggles and, thus, indicated to the user that "there is more."

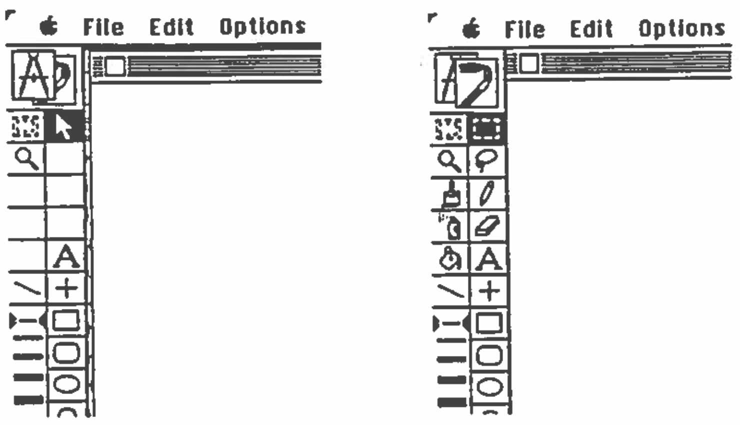


FIGURE 5. The SuperPaint Palette in Draw Mode and in Paint Mode (*a* portrays the Draw Mode; *b* portrays the Paint Mode)

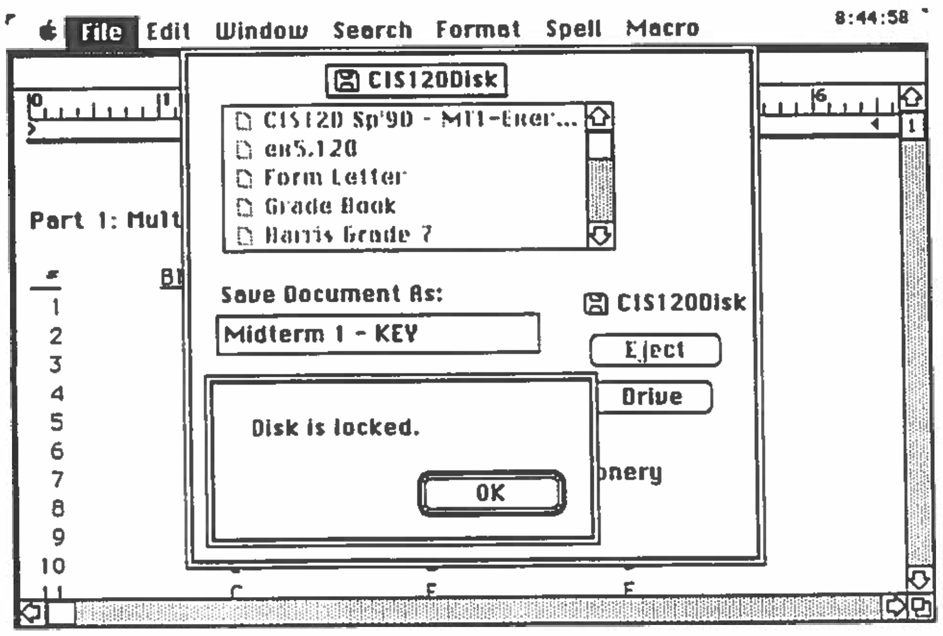
User Performance as a Basis for User–Interface Patterns (UIPs)

Another very effective method for creating UIPs is to study people using computers. Alexander (1979) used a similar method in his work. Some patterns may be developed based on knowledge about why a particular structure is unsuccessful. In other words, instead of attempting to determine which elements make a structure successful, it is often easier to determine which elements make it unsuccessful, and to then develop a pattern based on that knowledge. In the user–interface domain, a large number of problems arise during the interaction between humans and machines; and, many of these problems may be attributed to the design of the user interface. Some of the problems are obvious. For example, the user may not understand a command name; the user may not know how to move/size/close a window; the user may not know how to select a block of text; and so on. These problems all can be predicted without examining users. Other problems, however, are more subtle. During the course of this thesis study, a large number of less obvious difficulties have been observed. One needs only to consider the following three examples:

1. Many novices do not know how to respond to the many messages (or alert boxes) that often appear during the use of Macintosh applications. It has been observed that, after an alert box has been displayed, users often sit around waiting for it to disappear by itself. For example, when a disk–locked message is shown, novices often sit passively waiting for it to vanish (see Figure 6). The remarkable issue is that the same users have negligible problems terminating a Print command by clicking the *OK* button in the standard print window. It is intriguing that the problem arises in some instances, but not in others. The underlying reason seems to be twofold. The problem does not arise when a mouse click is required to terminate an action initiated explicitly by the user (e.g., Print and Save As). On the other hand, whenever an alert box is displayed (e.g., the disk–locked message) as a side effect caused by an action initiated by the user, then the problem occurs. Moreover, this difficulty does not seem to be as evident when the alert box contains a question instead of a statement.

2. Another type of problem that was observed among novice users appeared to stem from the fact that some commands do not provide sufficient feedback when they are carried out. Numerous examples exist, such as the Copy command which is available in most Macintosh applications. When selecting Copy, there is no indication of any change; many users selected Copy

repeatedly, expecting some visual feedback. Turning on the grid in MacPaint does not affect the screen; many users repeatedly selected the Grid command, but believed that they had done something wrong since no changes were visible. This problem is related to yet another third observation: Many menu items act as toggles. Some menu items are checkmarked to indicate that an item is selected, but others menu items simply change (e.g., the Turn Grid On item shifts to Turn Grid Off). These are all subtle changes that the first–time user cannot be expected to detect immediately; therefore, the amount of feedback is insufficient.



Save Document As:

Midterm 1 - KEY

Disk is locked.

FIGURE 6. Disk–Locked Message

3. An interesting problem arose when users were required to manage a relatively large number of windows on a small Macintosh screen. The problem

occurred during the use of Microsoft Works. The task was to integrate information from several databases, spreadsheets, and graphs into a single document. Naturally the screen became cluttered by the large number of windows; thus, some windows were often completely hidden behind other windows. The observed students worked with several such problems over a period of 5 weeks. It was discovered that they spent a significant amount of time moving and resizing windows in order to find hidden windows. They did not use the Window menu which could have saved time; although they were repeatedly reminded of this option, the behavior persisted. Even during the latter parts of the course, many students did not utilize the Window menu.

In the three preceding case examples, the problems arose because the conflicts among the different forces involved have not been resolved. The first case represents a conflict between the willingness of a novice to respond to events not initiated explicitly by him– or herself versus the large amount of information the system is programmed to convey to the user. A novice user is unprepared to deal with events that are initiated by the machine. In the second case, the conflicting forces partly arise from the differing needs of novice and expert users. Novices need a great deal of feedback when using a program; they must feel confident that the correct event is happening––and even that *something is*, in fact, happening. However, most programs are not written exclusively for novice users. Expert users do not need the same level of feedback. Hence, the conflicting requirements for feedback among the different user groups can be said to incorporate the conflicting forces in this case. The conflicting forces are far from obvious in the third case. The problem might be traced to the fact that dragging and resizing windows are some of the first actions a user is taught in regard to the Macintosh. Therefore, the move/size method might be the first method that comes to mind when a window must be uncovered. Another hypothesis might be that, in general, pulling down a menu encompasses more time than direct clicking. Furthermore, even though the Window menu is available, the name of a hidden window might be unknown (i.e., the user might have forgotten it); and, without the name of the desired window, the window menu is of little use. In general, considerable effort is required in order to obtain a good approximation regarding which conflicting forces may be causing a particular problem.

The Elements of a User–Interface Pattern (UIP)

The elements of a UIP are, in principle, the same as the elements of Alexander et al.'s (1977) patterns. In Chapter IV, the elements of each UIP are presented according to the following system:

1. An introductory paragraph is provided in order to set the context for the UIP. As previously discussed, there are both small and large UIPs, and the larger patterns need to be embellished by the smaller patterns. The larger patterns, which are embellish by the current pattern, are listed in the first paragraph (under the heading of Context). For example,

> The size the of computer screen always will be a limiting factor in terms of how much information can be displayed at any given time. This limitation must be minimized as much as possible. Users often want to view and compare pieces of information displayed in several windows; this potentially causes a large number of windows to be displayed on the screen all at one time. In addition, because the user wants to tailor the

screen layout to fit personal taste (through UIP 7: PERSONALIZABLE SCREEN LAYOUT), it is important to be able to manipulate windows in certain ways. (Chapter IV, pp. 81–82)

2. For some UIPs, a figure is provided in order to show an archetypical example of the UIP. At times a series of pictures are included to demonstrate the dynamics involved in the UIP (as is the case with UIP 12: MANIPULABLE WINDOWS).

3. The problem statement is a succinct specification of the problem at hand. For example, "The size of a computer screen will never be large enough to accommodate the user with respect to the information that needs to be displayed" (Chapter IV, p. 82). In some cases, the problem statement specifies the conflicting forces. However, in most cases, the numbers of conflicting forces are so large, or they are so complex, that they are not included in the problem statement; in such cases, they are addressed within the body of the UIP instead.

4. The body of the UIP discusses the conflicts and addresses the validity of the pattern. If there are several ways to include the pattern into a design, each method is specified. Examples that illustrate cases in which the pattern is not followed are presented when appropriate. Ideally, the validity of all statements that are made herein should be backed up by references to completed research. However, in this thesis, personal experience and ideas provided by other user–interface experts have been utilized. Not until the field of user–interface design matures can all the arguments refer to sound findings. (In addition, the goal of this thesis was to constitute a feasibility analysis, rather than to attempt the development of a complete pattern language.)

5. The solution is a succinct instruction regarding the construction of the UIP; or, in other words, it concerns resolving the conflicting forces. For example, under the heading Therefore, "The user should be able to size, open, close, move, and select the windows that display information. No arbitrary limit should be set regarding how many windows can be open at any given time" (Chapter IV, p. 82).

6. A reference to smaller UIPs specifies which UIPs embellish the current UIP, under the heading Embellishments; for example,

> There are often objects on the screen (e.g., palettes and menu bars) that need to be protected from being hidden by the windows that can be moved around on the screen (this is addressed by UIP 13: PROTECTED ELEMENTS.) Furthermore, this pattern should be embellished by UIP 25: ELEMENT BOUNDARIES in order to separate a window from other windows and other element types, and by a UIP that discusses how a menu (or other methods) can be used for window handling. Finally, patterns should be developed to address how the window manipulations should be carried out. (Chapter IV, pp. 82, 84)

The User–Interface Patterns (UIPs) as a Language

It has been demonstrated that it is possible to identify and create UIPs, and the different elements of a UIP have been defined. The UIPs that eventually may be discovered will cover a wide range of issues which are relevant in user–interface design. The large UIPs will encompass aspects of the overall interface structure. The midsize patterns will embellish the larger patterns, often by describing somewhat smaller problems that arise as a result of including the larger patterns. For example, there might be a UIP that addresses the use of palettes. Once a designer has made the decision to include a palette, a new problem arises; for instance, one must determine where the palette should be located on the screen. The smaller patterns deal with the more detailed aspects of user–interface design; for example, once a palette has been positioned on the screen, one must establish how the various choices in the palette should be separated and how the various options should be selected. Although this thesis has defined various large and midsize patterns, creating patterns that address detailed aspects of user–interface design not only is very difficult, but also is beyond the scope of this thesis.

A UIP is not an isolated entity; it is dependent upon other UIPs, and the latter UIPs depend on yet other UIPs. The structure of a pattern language follows from the interdependent nature of patterns. The example illustrated in Figure 7 conveys this idea. Several questions immediately arise: Why is it obvious that there is a functional palette somewhere on the screen? Why is it clear that the palette is indeed functioning as a palette? A partial answer is that the palette contains a related set of smaller UIPs. A palette is usually divided into small rectangular boxes, with each box possessing an icon and/or some command name; furthermore, in a palette, only one of the functions is active at any given time, and this active function is distinguishable from the others.

But the partial answer in the preceding paragraph is insufficient because it considers only the smaller patterns in order to define a palette. There most certainly is not a functional palette depicted on the screen in Figure 8. A palette must be defined in terms of larger patterns as well. Thus, a palette, as a collection of functions, is just one possible variation of a menu where different functions can be selected. The selection of functions usually transpires along the edges of the screen. Other activities, such as drawing and writing,

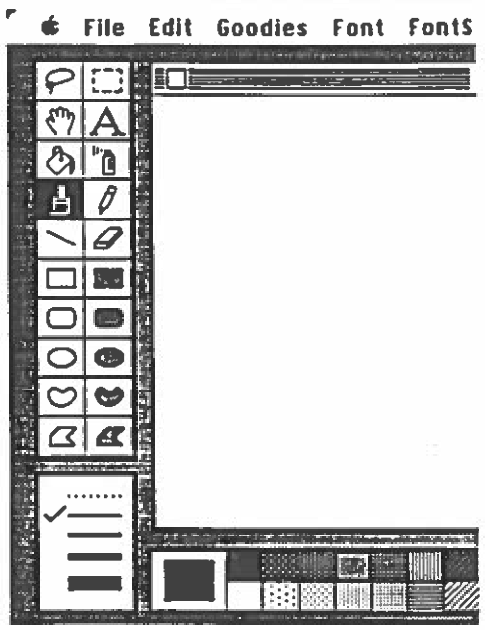


FIGURE 7. A Palette in a Usual Context

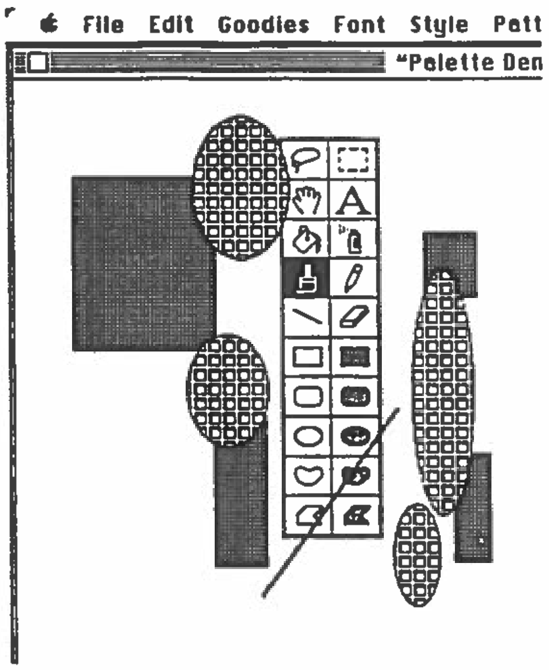


FIGURE 8. A Palette in an Unusual Context

occur closer to the center of the screen. Moreover, a palette usually does not cover other entities within the user interface, neither is it covered by other entities. (However, the new version of MacPaint uses a tear–off menu and allows the user to position the palette anywhere on the screen; although the palette may cover other elements, other entities are not allowed to cover the palette.)

The palette depends both on the smaller UIPs it contains and on the larger UIPs within which it is contained. This is true of all UIPs. Consider the menu bar which becomes merely an insignificant sequence of words if it is placed in a different context. The set of words become a menu bar only when the words are located on the top of the screen––typically with a different background color than the rest of the screen, or at least with some sort of boundary that separates the words in it from all the other interface elements. The menu bar is protected from being hidden by other elements (see Figure 9). Similarly, an icon is merely a small picture; it has a different meaning when it occurs inside a document window, than when it occurs in a window on the desktop (i.e., in the latter case, it represents a file or an application).

The preceding examples indicate that the relationships among various UIPs can be viewed as a network. Given a UIP *P*, there will be some smaller UIPs that embellish it, say *S–1* through *S–n*, and *P* may also be used to embellish some larger UIPs, say *L–1* through *L–m* (as illustrated in Figure 10). Figure 10 also illustrates that the larger patterns depend on other (midsize) patterns, and that the smaller patterns can be used to embellish patterns other than *P*.

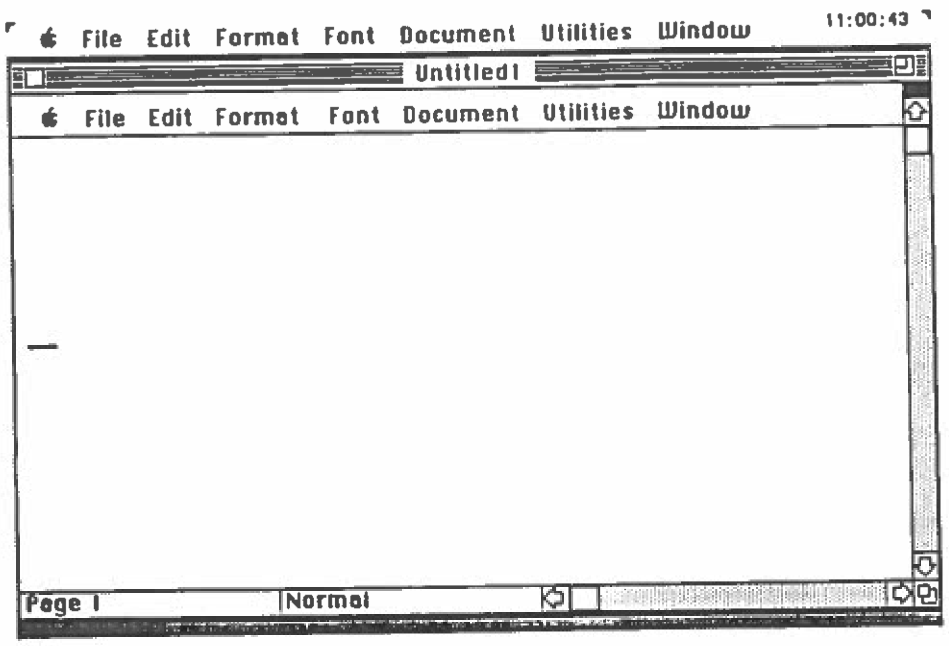


FIGURE 9. A Menu Bar and a Set of Words

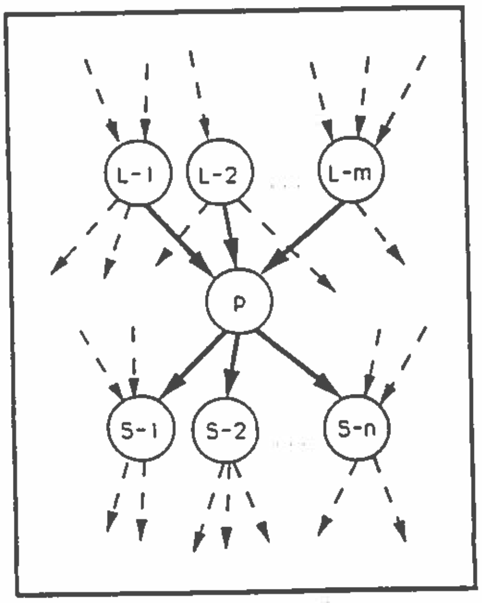


FIGURE 10. Many User–Interface Patterns (UIPs) Form a Network of User–Interface Patterns (UIPs)

Even though the UIPs form a network which constitutes a language, further questions arise: How can it be established that the language is complete? How can it be known whether some patterns should be left out or other patterns added to the language? When is a set of patterns sufficient for making an interface work as a whole? Alexander (1979) argues that a pattern language is complete if and only if it is both *morphologically* and *functionally* complete.

Morphological Completeness

A pattern language is morphologically complete when the UIPs form a complete interface within which all the details have been addressed. The language should answer all the questions that a designer might have about the interface. Suppose, for instance, that UIP 10: PALETTES is being considered (as is discussed further in Chapter V). Suppose further that there is no UIP that indicates where to position a palette and there is no UIP that indicates how a palette should be used with respect to function selection; that is, it is unknown whether or not more than one function, or mode, can be selected at a time. It, therefore, is not possible for the designer to visualize the use of a palette because of insufficient information; in other words, there is a significant gap in the understanding of the issues involved. A language is morphologically complete only when all relevant questions are answered by the UIPs in the language.

Functional Completeness

A language is functionally complete when the UIPs in the language resolve all the conflicting forces that arise. An interface which contains unresolved conflicting forces will result in an unsuccessful interface. The collection of UIPs must stabilize all the opposing forces.

Discovering Complete Patterns

In order for a language to be complete, each individual UIP must be complete. The sub–UIPs of a particular UIP must guarantee that this specific UIP is morphologically complete; and, by resolving the conflicting forces that this UIP generates, the sub–UIPs must also guarantee that this UIP is functionally complete. Therefore, it might become necessary to define additional UIPs as result of having defined a certain UIP. For example, once a pattern has been defined to introduce the notion of a palette, several new problems become important. Specifically, there probably will be some conflicts between the palette and the manipulable windows that are present on the screen. Many issues, such as the following, must be addressed by the smaller UIPs: Should the palette be protected from being ensconced by other screen elements? Where should the palette be positioned? What is the minimum/maximum number of functions that should be included in the palette?

In conclusion, a specific UIP is typically a module of some larger UIP, and the specific UIP exists as a result of the forces in the larger pattern that must be resolved. Similarly, when a new UIP is created, a need arises for smaller UIPs to resolve the forces generated by the new pattern. There is a considerable amount of work involved in preparing a pattern language; and it is this language which determines the finished design. The language should completely specify the interface being designed.

Summary

This chapter has explicated the distinction between identifying and creating patterns; it also has provided examples of how patterns can be created. Although some of the UIPs that were developed as a part of this thesis document simply have been identified, most were created based on problems observed among computer users. Human psychology and user performance, as previously discussed, form the bases for most of the 30 UIPs which are presented in Chapter IV. A set of UIPs form a network in that a UIP depends on both smaller and larger UIPs. The notions of morphological and functional completeness was introduced to emphasize that a pattern language is morphologically complete only if all the details of a particular user–interface design project are addressed by the patterns. In other words, there must be a complete translation from system requirements and functional semantics to a user–interface design specification. There must not be any "holes" in the design specification. In addition, a pattern language is functionally complete only if it resolves all the conflicting forces that have been identified. (The reader also has been referred to Appendix A for an example of how an architectural pattern may be created to satisfy a set of requirements.)

CHAPTER IV

USER–INTERFACE PATTERNS

Overview

This chapter presents all the UIPs that were identified over the course of this study. Some user–interface patterns (UIPs) will undoubtedly seem obvious, depending upon the background and experience of specific readers; nevertheless, other UIPs will offer new ideas. Even within the architectural domain there were the apparently *obvious* patterns (such as portrayed in Alexander et al.'s, 1977, pattern entitled SOUTH FACING OUTDOORS).

Some of the solutions proposed to be parts of the larger UIPs may seem just as vague as the principles stated in traditional user–interface guidelines. In such cases, the sub–UIPs will fill in the details and provide specific instructions regarding how the larger UIP can be satisfied. After the patterns have been defined, an example that demonstrates how they can be used to design a user interface is presented.

Thirty Specific User–Interface Patterns

UIP 1: INDEPENDENT PROGRAMS

Context

Once a decision has been made to write a new program, the programs that the new program will depend on must be taken into consideration. For example, most application programs assume the existence of (as well as depend on) an operating system. Dependencies upon the operating system or any other program must be hidden from the user. For example, if a program requires a particular setting of operating system–level parameters, then the program (not the user) should set these parameters.

Problem

A program will be of little or no use unless it can be fully utilized without the user worrying about its dependencies upon other programs.

Discussion

Learning how to use a program is typically a nontrivial task. A novice would find this effort even more laborious if the system's dependencies upon other programs were not hidden. The user should not have to learn how to use Program A in order to use Program B. In practice, however, it is close to impossible to avoid this altogether. Consider, for example, the relationship between an operating system and the applications that run under the operating system. The user typically must learn the operating system–level commands before running the application. In other words, the user has to learn *A* before learning *B*. However, by limiting the number of commands that need to be mastered, and by carefully choosing how the commands are to be executed, the learning time of other programs can be reduced significantly.

Therefore

Effort should be directed towards developing programs that do not require prior knowledge of other programs. The fact that Program A depends on Program B should not require the user to learn Program B. The dependencies should be hidden by the system.

Embellishments

On all systems, designate an UIP 2: ACTIVITY CENTER within which all the independent programs are managed. If it is possible within the programming community,[1] select a few user–interface elements (such as through UIP 3: STANDARD SET OF ELEMENTS) upon which all user interfaces within the community can be based. At the same time, ensure that each program has a UIP 4: DISTINGUISHING FEATURES. Enable and encourage the sharing of data between the independent programs using UIP 5: TRANSFER OF DATA, and ensure UIP 6: EASY TRANSITION BETWEEN PROGRAMS.

UIP 2: ACTIVITY CENTER

Context

For most systems, a wide array of programs are available; and for many users, several of these programs will be of relevance. In order to complete

[1]The term *programming community* will often be used in the UIP definitions. A programming community is simply a set of companies and/or individuals that write software for a specific type of machine or machines.

UIP 1: INDEPENDENT PROGRAMS, a decision must be made regarding how all these programs are to be managed. Figure 11 demonstrates one such possibility.

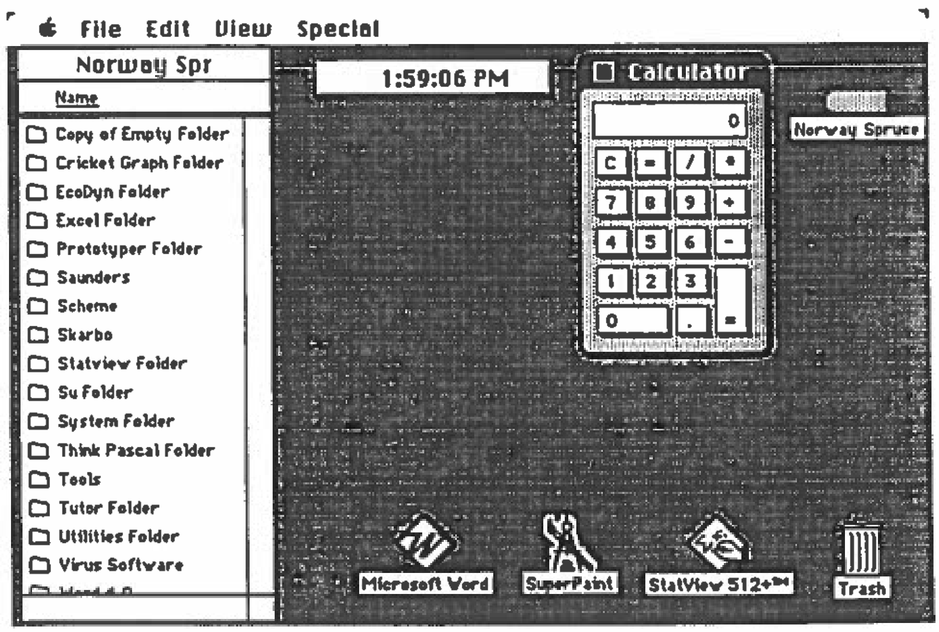


FIGURE 11. An Example of an Activity Center

Problem

The user needs a place at which all high–level activities concerning the organization and the manipulation of all the data and available programs can take place.

Discussion

This pattern is based on how people orient themselves in their surroundings. People use mental maps, and mental maps require points of reference (Johnson–Laird, 1983). For example, when considering a complex of buildings, the point of reference probably would be a building placed in such a manner that all references to other paths and buildings could be made using this point of reference (the first building) as a basis. An analogous situation exists in the domain of user–interface design. There must be a point of reference––a place or a program––which acts as the functional soul of the entire system. In Figure 12, Program P acts as a central point of reference for all other activities.

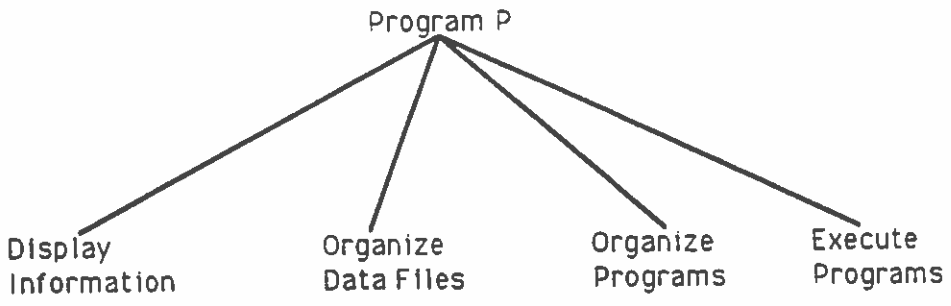


FIGURE 12. The Activity Center as Central Point of Reference

<u>Therefore</u>

On any system, a program should be included that acts as the central point for all high–level activities (such as organizing application programs, organizing data files, and executing programs).

<u>Embellishments</u>

The activity center should be provided with a facade (such as in terms of UIP 4: DISTINGUISHING FEATURES) that is different from all other programs available on the system. The user should be allowed to tailor the layout of the elements to accommodate personal preferences; that could be accomplished through UIP 7: PERSONALIZABLE SCREEN LAYOUT. In addition, limit the number and complexity of the commands (through UIP 8: LIMITED NUMBER OF COMMANDS) so that the user does not have to spend a considerable amount of time learning how to use the activity center prior to learning how to use the applications of interest.

UIP 3: STANDARD SET OF ELEMENTS

Context

When it has been established that more than one program will be used by a single user (UIP 1: INDEPENDENT PROGRAMS), then steps must be taken to promote learning by the transfer of knowledge.

Problem

People feel comfortable in familiar environments; and people often feel uncomfortable in unfamiliar environments. The ease of transition from one environment to another is directly related to the sameness of the two environments.

Discussion

It is interesting to examine the difficulties and inner conflicts that a person encounters when the environment changes. Consider, for example, moving from City A to City B. If City A and City B are neighboring cities, the environment scarcely changes. However, if City B is located in a different state, the environment may be somewhat more different. And, if City B is in a different country, the environments are typically significantly different. The more different City B is from City A, the harder it is to adjust to City B.

A similar situation exists in the domain of user interfaces. If a user has been using Program X for some time and needs to use a different program (Program Y), then the ease of transition from Program X to Program Y is directly related to the sameness of Program X and Program Y.

Within a programming community, a limited set of user–interface elements should be defined. The point is that, if the programs have the same functionality, then they should use the same set of user–interface elements. No additional elements other than those elements from that set should be included in the user interface. This does not mean that all user interfaces will be almost identical in appearance. The elements can be combined in a practically infinite number of ways. For example, most Macintosh user interfaces are based on a small set of elements (including windows, icons, menus, menu items, radio buttons, check boxes, and controls). Some of these elements can be further refined; for example, there are several window types.

Therefore

The transfer of knowledge should be promoted by defining a limited set of user–interface elements (as discussed in the preceding). All user interfaces within a programming community should use only these elements.

Embellishments

Most user–interface elements have some functionality attached. Consequently, not only should the user–interface elements look the same, but also they should work the same. Given the power to enforce it, include UIP 9: STANDARDIZED METHODS; and define standard elements that can be used as vehicles for communication between user and machine. Consider UIP 10: PALETTES, UIP 11: PULL–DOWN MENUS, and UIP 12: MANIPULABLE WINDOWS. (The reader should note that, although there are other standard elements to consider, they are beyond the scope of this thesis.)

UIP 4: DISTINGUISHING FEATURES

Context

Once it has been decided to include UIP 2: ACTIVITY CENTER, some method to establish the center as unique must be designed. The user must be able to distinguish the center from all the other programs that can be accessed. In general, each program within a programming community should have some user–interface element that distinguishes it from all other programs. In addition, there should be a UIP 6: EASY TRANSITION BETWEEN PROGRAMS; this implies that, once the center of activity (or some other program) has been accessed, the user should immediately recognize that a transition has occurred based on some physical features in the interface. At any given time there should be no doubt as to which program is currently active. In other words, there should be no confusion as to which mode is currently active.

Problem

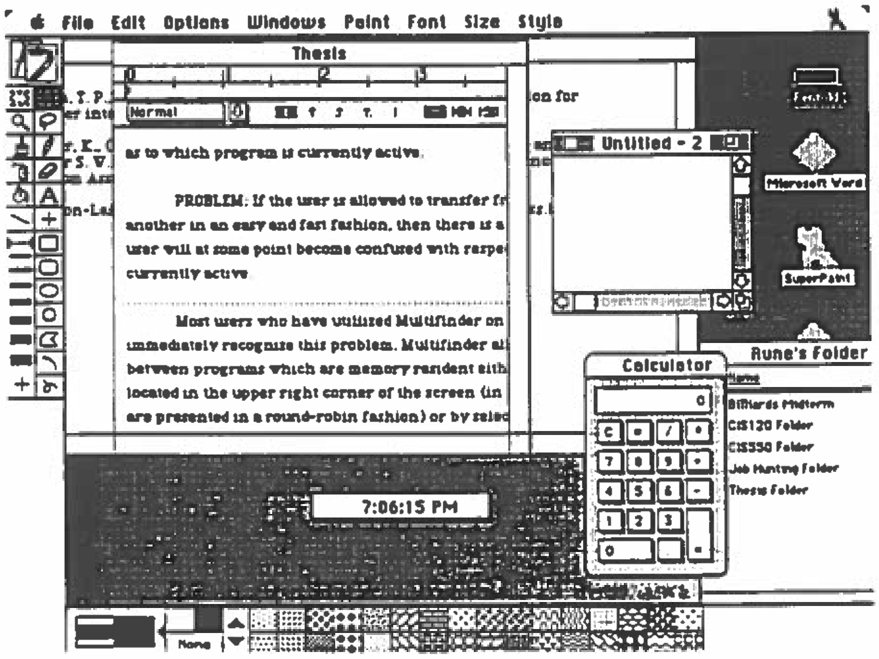
If the user is allowed to transfer from one program to another in an easy and rapid manner, then there is a high probability that the user will at some point become confused with respect to which program is currently active. In general, mode changes can lead to confusion as to which mode is currently active.

Discussion

Most users who have utilized MultiFinder on a Macintosh will immediately recognize this problem. MultiFinder allows users to easily switch from one memory–resident program to another either by clicking on an icon located in the upper–right corner of the screen (in which case the programs are presented in a round–robin fashion), by selecting a program from the Apple menu, or by clicking on an interface element belonging to the target program. In any case, the screen is often very cluttered, and it is often difficult to determine the screen elements that are part of the selected program (see Figure 13).

Although the name of this pattern is UIP 4: DISTINGUISHING FEATURES, this pattern has several purposes.

First, every program should include at least one element that makes the program easy to distinguish from all other programs. To be creative in this respect is not difficult. The possibilities are virtually infinite in number. For example, the center of activity on the Macintosh is the Finder which uses the desktop metaphor. There are several elements that assist in distinguishing the desktop from any other program. For example, the trash–can icon is always present, and at least one icon that represents a disk is constantly displayed. What are effective and ineffective distinguishing features? Is a menu bar effective as a distinguishing feature? Clearly not. Many menu bars are often very similar, and there is more time involved in examining a menu bar than there is in locating a graphical element on the screen. A user–interface designer will know when a good distinguishing feature has been found. Other distinguishing features might include palettes, rulers, grids, and background color. If in doubt, an experiment that measures the time it takes for the subjects to identify different programs can be conducted.



as to which program is currently active.

PROBLEM: If the user is allowed to transfer fr
another in an easy and fast fashion, then there is a
user will at some point become confused with respe
currently active.

Most users who have utilized Multifinder on
immediately recognize this problem. Multifinder al
between programs which are memory resident eith
located in the upper right corner of the screen (in
are presented in a round-robin fashion) or by selec

FIGURE 13. Several Programs Being Managed by MultiFinder

Second, this pattern addresses some of the difficulties induced by MultiFinder. It is not sufficient with a few distinguishing features if MultiFinder is "abused." In Figure 13, several distinguishing features are visible, but they convey little information as to which program is active. However, all the blame should not be attributed to MultiFinder. The problem lies in the fact that most programs do not explicitly clear the screen when they are accessed via MultiFinder. One that does clear the screen is MacPaint; thus, one solution is simply to clear the screen when changing program context. The conflicting forces that arise as a result of this solution are due to the fact that some users prefer to activate programs by clicking on visible elements belonging to the target program. One way to please these users is to *gray out* the areas that are not active. Figure 14 depicts a potential pattern that might be called *Background Inactive Areas* (which is not defined in this thesis). The success of this approach may be questioned; it has not been empirically tested whether graying out inactive elements will reduce the difficulties discussed. (The reader should notice that the desk accessories in Figure 14 are not grayed out.)

Therefore

Elements always should be included that help distinguish a program from other programs. Such elements may include special icons, palettes, rulers, grid lines, and background. Moreover, when one program is accessed from another either (a) erase all the elements on the screen before the new program is activated, or (b) gray out the elements that do not belong to the newly activated program.

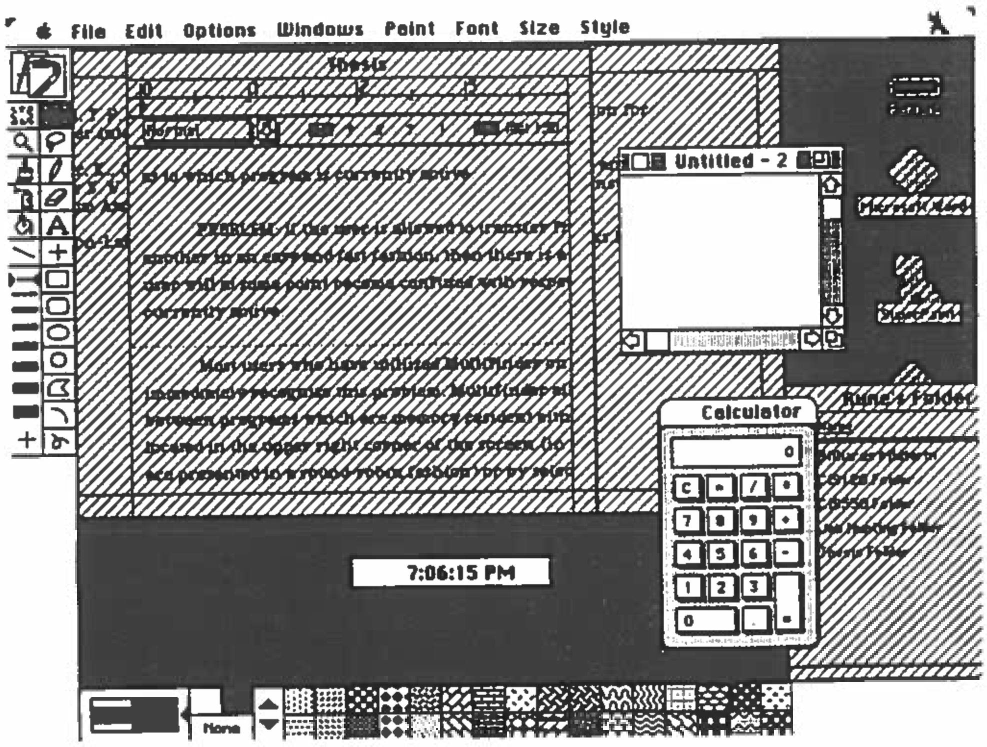


FIGURE 14. Graying Out Inactive User–Interface Elements

Embellishments

It is often necessary to protect certain elements of an interface from being moved, hidden, and/or destroyed in any way. In order to complete this UIP, the distinguishing features must be protected (through UIP 13: PROTECTED ELEMENTS). If they are not protected, they will not effectively serve their purpose. Furthermore, the layout of the elements of a program, or the state of a program, should not change while the user is accessing a different program; this refers to UIP 14: SAVING THE PROGRAM STATE. UIP 10: PALETTES should also be considered as a distinguishing feature.

UIP 5: TRANSFER OF DATA

Context

In order to complete UIP 1: INDEPENDENT PROGRAMS, methods that will allow the transfer of data from one program to another must be included.

Problem

If a program is built independently from all other programs, then methods for transferring data from one program to another often will be overlooked.

Discussion

In UIP 1: INDEPENDENT PROGRAMS, it was argued that all programs should be developed on an individual basis; this means that Application A should not depend on Application B. Unfortunately, this may undermine the sharing of data between different programs. For example, even though a statistics program may serve its purpose by allowing the user to directly enter data, the program will cause considerable irritation unless it also allows the user to import data produced by other programs (such as, e.g., from a spreadsheet program).

It might be questionable whether or not it makes sense to import pictures into a spreadsheet program. However, does it make sense to import pictures into a word–processing program? The answer to the latter question is undoubtedly yes. Why should not the answer to the first question also be yes? It might be argued that users should be allowed to import pictures into a statistics program. One benefit of this would be that the user could enhance the graphical descriptions of data with informative illustrations. It is not without reason that some statistics and graph programs have a functionality that allows the user to make simple drawings that accent the output.

Therefore

The user should be allowed to transfer data from one program to another. This may be accomplished by including a communication channel that is recognized by all the programs within the programming community. If necessary, the receiving program may handle the received data differently than the way it handles data normally; then the receiving program would be responsible for any necessary reformatting of the imported data.

Embellishments

In order to implement the capability of transferring data, functionality should be included for selecting data (UIP 15: DATA SELECTION) and inserting data (UIP 16: DATA INSERTION). Because data have to be transferred from one program to another, a means needs to be provided for temporary storage of the data to be transferred (such as through UIP 17: SHAREABLE SCRAPBOOK). Finally, the transition between different programs should be easy and fast (UIP 6: EASY TRANSITION BETWEEN PROGRAMS).

UIP 6: EASY TRANSITION BETWEEN PROGRAMS

Context

In order to complete UIP 1: INDEPENDENT PROGRAMS (and UIP 2: ACTIVITY CENTER), alternative means for fast and easy transitions between programs must be provided. The user should not be required to enter the activity center prior to launching a program; instead, the user should be able to take shortcuts. In addition, UIP 5: TRANSFER OF DATA between different programs should be just as fast as moving data from one place within a document to another place within the same document.

Problem

Users will avoid switching between programs unless absolutely necessary if the transition is slow and cumbersome.

Discussion

Users who want to exit Program A and then execute Program B often have to exit Program A, enter the system's activity center, and then access Program B. If the user needs to merge data created by several different programs, this is particularly annoying. For example, in the process of writing this document, the figures were initially created once the need for them arose. This involved many transitions between Microsoft Word, MacPaint, and MacDraw. However, because none of these programs offer the user the capability to directly launch another program, the activity center had to be entered between each transition. (MultiFinder was unavailable due to memory limitations.) Consequently, the preferred method for creating a document was abandoned because the transition between programs was painstakingly cumbersome.

Therefore

All programs should include a standard method for transferring directly from one program to another. This can be done in at least two ways (discounting utilities such as MultiFinder):

1. Within each programming community, provide a standard utility (such as Switcher) to manage the transitions between programs that are relevant. It should not be a necessary requirement for all programs to be memory resident at the same time.

2. Include a Transfer... option in all programs.

Embellishments

In order to complete this pattern, the state of the abandoned program must be saved (e.g., through UIP 14: SAVING THE PROGRAM STATE.)

UIP 7: PERSONALIZABLE SCREEN LAYOUT

Context

UIP 2: ACTIVITY CENTER is of great importance to most users. This is where programs, files, folders, and utilities (e.g., desk accessories) are presented and accessed. In order to complete UIP 2: ACTIVITY CENTER, users should be allowed to tailor the screen layout to fit personal needs (see Figure 15).

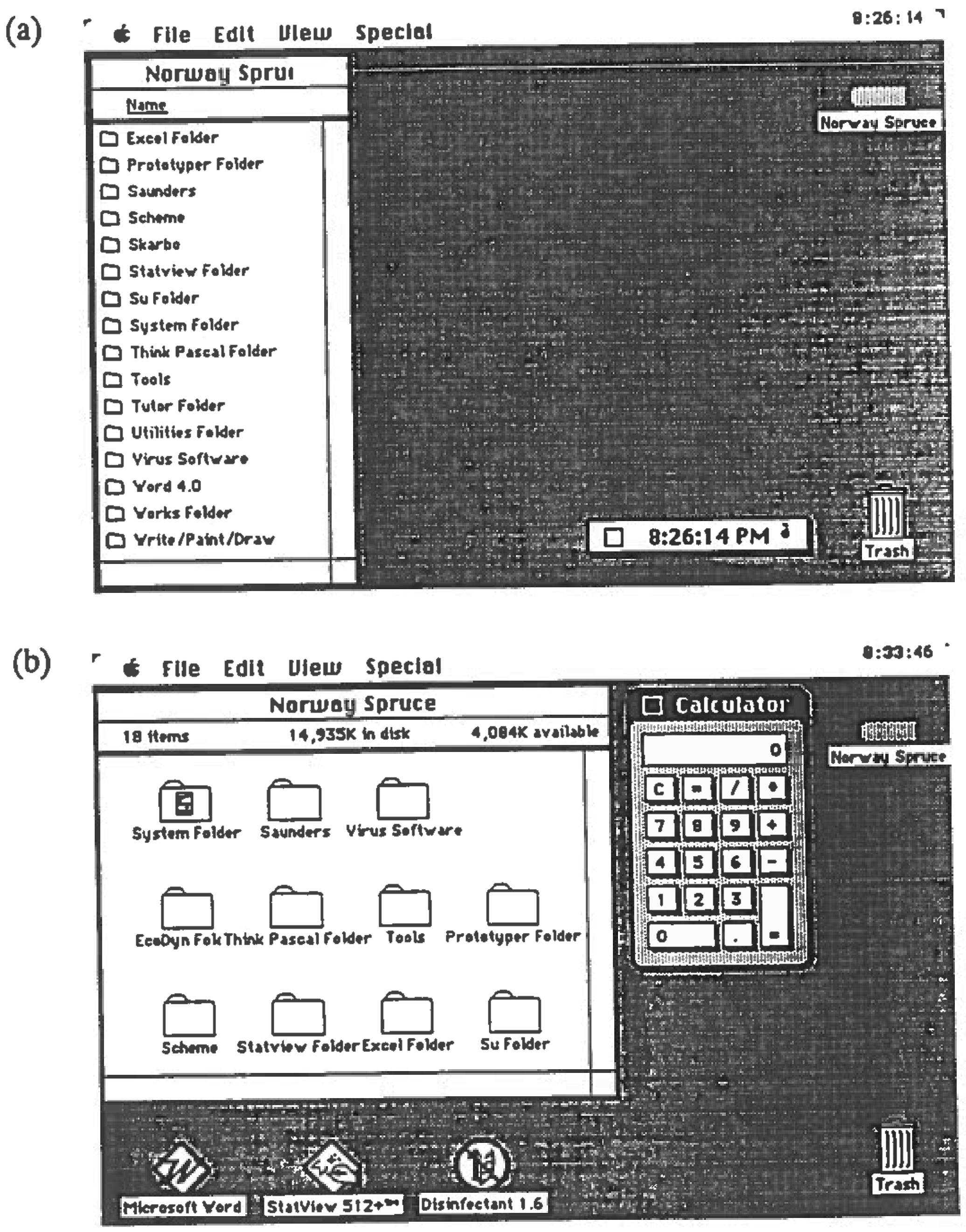


FIGURE 15. Two of Many Ways to Organize a Center of Activity (*a* portrays a desktop with all folders listed alphabetically; *b* portrays a desktop with all folders represented by icons, and with the most frequently used programs easily accessible)

Problem

The user never will feel comfortable with a user interface unless the elements can be displayed in a manner that satisfies the user.

Discussion

In the domain of user interfaces, since people are all different, there will be an infinite set of preferences with respect to (a) how information should be displayed on the screen, (b) when information should be displayed, and (c) where information should be displayed. Consequently, the user must be allowed to tailor the user interface––at least with respect to these three parameters.

Therefore

The users should be allowed to move, size, position, display, and hide the various elements of the user interface in order to accommodate individual preferences.

Embellishments

Even though the user should be given virtually unlimited freedom with respect to the how and which elements are displayed, some elements must be protected (such as those described within UIP 13: PROTECTED ELEMENTS). The user not only should be able to tailor the layout and choice of interface elements, but also should be able to determine the level of detail at which data are displayed (e.g., through UIP 18: LEVELS OF VIEW). Finally, in order to make this pattern useful, the system must automatically save the preferences specified by the user (through UIP 14: SAVING THE PROGRAM STATE).

UIP 8: LIMITED NUMBER OF COMMANDS

Context

If a project involves designing the UIP 2: ACTIVITY CENTER, the complexity of the center must be limited to a small fraction of the complexity of the programs that will be accessed from the activity center.

Problem

Users will become frustrated if they are required to spend considerable effort learning aspects of a system which are unrelated to the specific programs that they actually want to learn.

Discussion

One reason the Macintosh has been called user friendly compared to IBM machines can be traced to the differing pieces of software that implement the UIP 2: ACTIVITY CENTER. The number of and the syntactic complexity of the commands and actions that are available in the Macintosh activity center (the desktop) is much less compared to the commands available in the IBM activity center. For example, to copy a file from one disk to another on the Macintosh, the user *drags* an icon representing the file onto the target disk. In a typical IBM activity center, copy action has to be literally spelled out; spelling errors become an important issue. The Macintosh relies on command recognition rather than command recall. However, this comparison may be somewhat unfair because the IBM gives more power to expert users.

The most important factor that has made the Macintosh user friendly is the command syntax. Instead of requiring the user to type in commands via the keyboard, the user selects objects using a mouse. This technique is called *direct manipulation* (Hutchins, Hollan, & Norman, 1986).

Therefore

The number and complexity of activity–center commands available to the user should be limited to (a) COPY, for copying data from one storage medium to another; (b) MOVE, for moving data from one storage medium to another; (c) DELETE, for deleting data from a storage medium; (d) CHANGE DIRECTORY, and (e) EXECUTE, for running programs. In addition, commands required for the patterns described in the next paragraph below must be included. Finally, command recognition should be relied on (rather than command recall) by arranging the commands in palettes and pull–down menus.

Embellishments

This UIP is of little or no use unless the user is given immediate feedback regarding the actions that are executed (e.g., through UIP 19: IMMEDIATE FEEDBACK). Unfortunately, although all actions should be reversible (e.g., through UIP 20: REVERSIBLE ACTIONS), some actions are practically impossible to reverse. (Consult UIP 21: WARNING THE USER for an in–depth description of this problem.) Within a programming community, standard methods must be developed to select and insert data (such as UIP 15:

DATA SELECTION and UIP 16: DATA INSERTION). The user should be allowed to view data at different levels of detail (i.e., with UIP 18: LEVELS OF VIEW). Finally, UIPs should be developed to address direct manipulation, command recognition instead of command recall, menus, and menu items. This UIP 8: LIMITED NUMBER OF COMMANDS will not be complete until these UIPs exist.

UIP 9: STANDARDIZED METHODS

Context

Most programmers do not have control over decisions that affect a vast number of programs. However, a programming community can benefit from identifying all the functions and options that the programs within the community have in common. For example, to complete UIP 6: EASY TRANSITION BETWEEN PROGRAMS, the programming community must agree on a standard way to accomplish the transition.

Problem

Users cannot utilize previous knowledge in the process of learning new programs unless the programs use the same means to accomplish the same goals.

Discussion

The assumption behind this pattern is that either we have the power to define user–interface style standards within our programming community, or we have a set of such standards available. If the goal is to make programs easy to learn, then the number of functions that the user has to learn should be limited. This does not imply that the number of commands available in a certain application should be limited; instead, this refers to the fact that available standards should be followed. In this manner, learning will be promoted by the transfer of knowledge, and the number of functions that needs to be learned from scratch will be limited.

Within the Macintosh community, the functions About..., New, Open, Close, Save, Save As..., Print, Quit, Cut, Copy, Paste, and Clear are included in most programs. Other aspects--such as text selection, dragging, mouse clicking, and user-interface element appearance--have also been standardized to some degree. There are also additional functions that are common to large subsets of the programs within a community. For example, there exists a large number of object-oriented drawing programs (MacDraw, MacDraft, SuperPaint, and CricketDraw). All these programs have commands in common such as Rotate, Group, Ungroup, Fill, and more.

However, one disadvantage of identifying user-interface standards, is that bad decisions, or bad standards, are difficult to change.

Therefore

Within a programming community, the syntax of the functions and actions that are common to all programs within the community should be standardized. The performance of the standard functions should be maximized by conducting empirical tests involving users. Updating of the standards should occur only if the new methods significantly increase overall performance; the programming community itself must decide whether or not the improvement is significant by weighing the cost of change against the benefits of change.

Embellishments

Currently none.

UIP 10: PALETTES

Context

Once the power to define a standard set of user–interface elements has been established (through UIP 3: STANDARD SET OF ELEMENTS), the palette may be considered as a good candidate for a standard user–interface element and as one of the UIP 4: DISTINGUISHING FEATURES.

Problem

Many tasks require frequent use of a limited number of different tools. Performing these tasks will lead to frustration unless the required tools are easy to access.

Discussion

A carpenter usually carries his or her hammer, nails, and tape measure in a utility belt. These are the tools that are most frequently used. For infrequent tasks, the carpenter fetches the needed tools from a toolbox. The carpenter knows which tasks are most common, and that there is a limit to how many tools can be carried around at all times. Furthermore, the carpenter knows that it is unwise to carry very heavy tools. The selection of tools that the carpenter keeps in the utility belt reflects this knowledge and these constraints.

There is an analogous situation in the domain of user interfaces. Some tasks require frequent mode changes or frequent function selections. Once again, MacPaint may be a useful example. The palette contains the most frequently used functions. In addition, it is fairly obvious which tool is active at any given time. There are several alternatives to a palette; for example, there is the alternative of placing the palette in a pull–down menu. An advantage is that screen space can be used for other purposes. Yet, there are several disadvantages. First, selecting a function is more time–consuming. Second, it may not be apparent which function is currently selected. (One way to eliminate this problem is to use different cursor shapes; unfortunately, this would require the user to learn mapping between cursor shapes and functions.)

<u>Therefore</u>

For every program, empirical studies should be carried out to identify the most frequently used functions. Then the most frequently used functions should be placed in palettes in order to reduce function–selection time.

<u>Embellishments</u>

The inclusion of this UIP leads to several difficult questions. What if the functions that are most frequently used are not related in any fashion? The functions in the MacPaint palette are closely related; but it may not make sense to place unrelated functions in the same palette. And, what if the set of most frequently used functions is task dependent? Easy solutions to these problems do not exist, but the use of *tear–off menus* may constitute a partial solution. A tear–off menu is a menu, generally graphic rather than textual, that the user can tear from the menu bar and move around the screen like a window. "Tear–off menus save desktop space, allow larger windows, and give the user more flexibility than do fixed palettes" (ACI, 1987, p. 91). If there are too many functions available in a palette, then the various functions become increasingly difficult to locate. One study suggests that the maximum number of codes for close to error–free recognition ranges from 5 to 10, for geometric and pictorial shapes, respectively (Grether & Baker, 1972).

If a palette is included in a user interface, then UIP 13: PROTECTED ELEMENTS must be considered––especially if the palette is used as a distinguishing feature (according to UIP 4: DISTINGUISHING FEATURES). UIP 10: PALETTES should be used in conjunction with UIP 11: PULL–DOWN MENUS. The selections made from a palette should be carried out exclusively and deliberately by the user. In no instance should the system alter a choice made by the user (i.e., UIP 23: USER IN CONTROL). Furthermore, this pattern should be embellished with UIP 25: ELEMENT BOUNDARIES in order to make the palette distinguishable from other elements. Finally, attention needs to be directed towards the patterns that address issues related to how it can be made clear to the user which functions are currently selected.

UIP 11: PULL–DOWN MENUS

Context

Once the power to define a standard set of user–interface elements has been established (i.e., through UIP 3: STANDARD SET OF ELEMENTS), and the fact that screen size is a limiting factor has been acknowledged, means by which the user can easily access all the available functions without sacrificing screen realty must be developed.

Problem

Novice users are better off if they do not have to remember command syntax; thus, users should be able to rely on recognition rather than recall. But if all available options are constantly displayed, there will be no room left on the screen.

Therefore

Group functionally related options (or menu items) into a maximum of 8 to 9 groups or menus. The user should be allowed to inspect one menu at a time; thus, the number of options that needs to be examined is limited. Place the menu name (e.g., File and Edit on the Macintosh) towards the edges of the screen.

Embellishments

This pattern is very vague, and there are many conflicting forces that remain unresolved. Given a large set of options, it is by no means trivial to break them into sets of related options. It is also difficult to decide how to order and group the items within a single menu. For example, should menu items that behave in a certain manner be placed directly above or below a menu item that behaves quite differently? Consider the toggles for bold, italic, underline, outline, and shadow in Microsoft Word. Each of these options work independent of the other options; selecting bold does not affect the status of underline. However, these toggles are grouped together with the plain–text option, which greatly affects the other options when selected. There are indications that such differences and inconsistencies add to the confusion experienced by novices when learning a system.

This pattern does not eliminate the need to remember command syntax. For example, in many cases, the user must remember that an object must be selected before a function is selected; however, this problem can be reduced with UIP 24: EXPLAINING MESSAGES. The selections made from a pull–down menu should be carried out exclusively and deliberately by the user. In no instance should the system alter a choice made by the user (e.g., UIP 23: USER IN CONTROL is active). Furthermore, this pattern should be embellished with UIP 25: ELEMENT BOUNDARIES in order to make a menu bar distinguishable from other element types. And, finally, this pattern should be complemented with UIP 10: PALETTES.

UIP 12: MANIPULABLE WINDOWS

Context

The size of the computer screen always will be a limiting factor in terms of how much information can be displayed at any given time. This limitation must be minimized as much as possible. Users often want to view and compare pieces of information displayed in several windows; this potentially causes a large number of windows to be displayed on the screen all at one time. In addition, because the user wants to tailor the screen layout to fit personal taste (through UIP 7: PERSONALIZABLE SCREEN LAYOUT), it is important to be able to manipulate windows in certain ways (see Figure 16).

Problem

The size of a computer screen will never be large enough to accommodate the user with respect to the information that needs to be displayed.

Discussion

Currently none.

Therefore

The user should be able to size, open, close, move, and select the windows that display information. No arbitrary limit should be set regarding how many windows can be open at any given time.

Embellishments

There are often objects on the screen (e.g., palettes and menu bars) that need to be protected from being hidden by the windows that can be moved around on the screen (this is addressed by UIP 13: PROTECTED ELEMENTS). Furthermore, this pattern should be embellished by UIP 25:

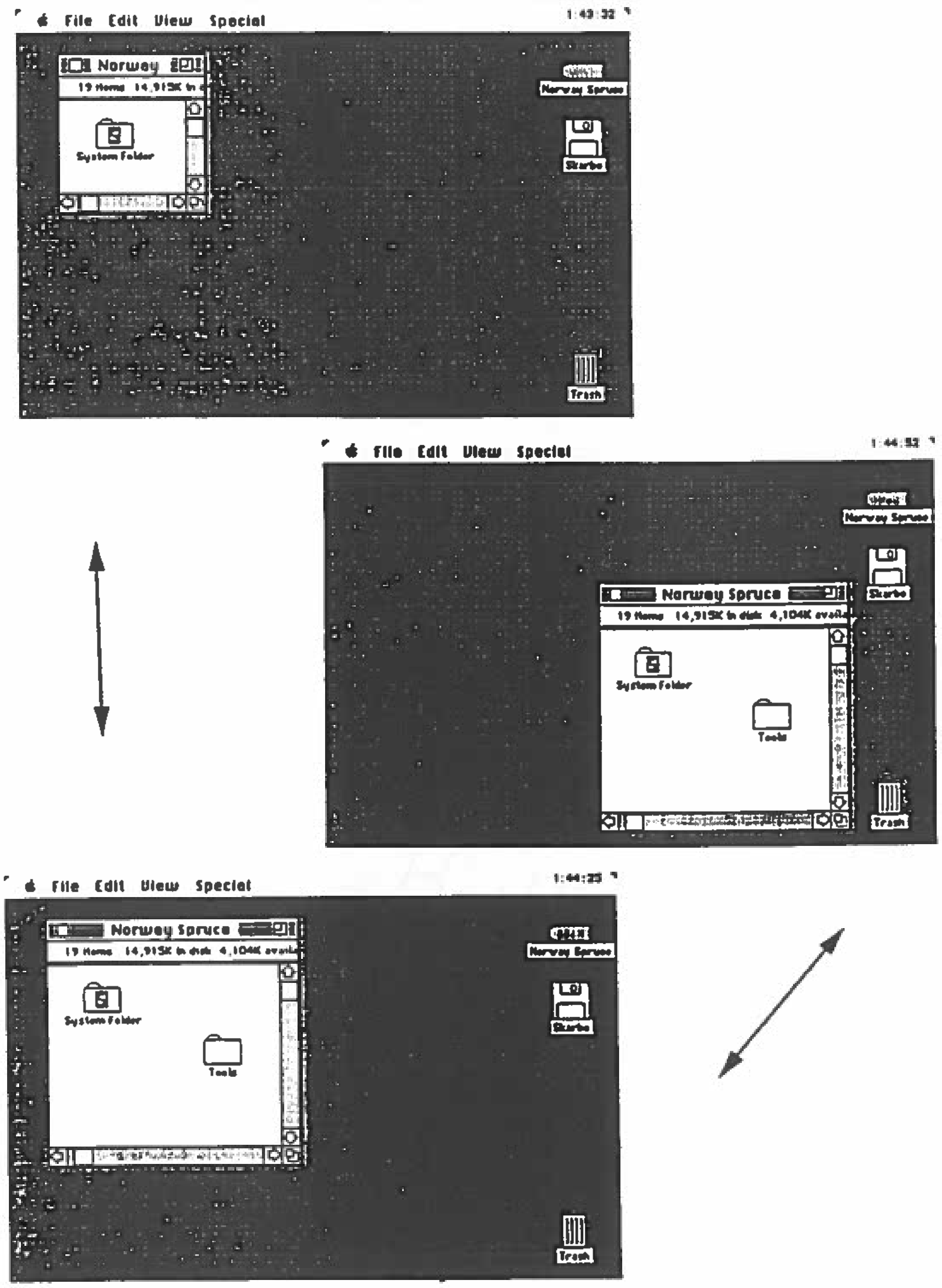


FIGURE 16. Manipulable Windows

ELEMENT BOUNDARIES in order to separate a window from other windows and other element types, and by a UIP that discusses how a menu (or other methods) can be used for window handling. Finally, patterns should be developed to address how the window manipulations should be carried out.

UIP 13: PROTECTED ELEMENTS

Context

After the distinguishing features of a program have been determined (e.g., through UIP 4: DISTINGUISHING FEATURES), methods for protecting these features (or elements) from ever being obstructed or hidden by other elements on the screen must be included. In particular, if the user is allowed to rearrange the interface elements (e.g., through UIP 7: PERSONALIZABLE SCREEN LAYOUT and/or UIP 13: MANIPULABLE WINDOWS), then steps must be taken to protect the distinguishing features and the elements from which the functions are selected (i.e., UIP 10: PALETTES and UIP 11: PULL–DOWN MENUS).

Problem

Giving the user total freedom with respect to how the elements on the screen may be arranged can quickly result in chaos. Moreover, some of the screen elements are sacred in that they are essential to the identity and functionality of a program.

Discussion

In UIP 4: DISTINGUISHING FEATURES, the importance of providing each program with its own identity (by including certain elements in the user interface) was addressed. Typically, these features can be tied together with the functionality of the program (such as, for instance, with a palette or a ruler). In general, all user–interface elements from which the user can select a function should be protected in some way or another. For example, the menu bar should be a protected element.

The reason such elements should be protected is due to the fact that they are so essential to the use and identity of a program that users should be prevented from, for example, moving windows on top of these elements and, thus, hiding them. This is particularly important if users frequently switch between many different programs. In addition, because the elements are often crucial to the functionality of the program, they must be protected so that the user can easily access the controls for the program.

Therefore

The distinguishing elements and any user–interface elements crucial to function selection must be protected. This may be accomplished by (a) ignoring all user actions that attempt to cover such elements by other elements; (b) disallowing the user to delete such elements from the screen; and (c) disallowing the user to move a crucial element on top of other elements that later can be selected and which, consequently, hides the crucial element.

Embellishments

This pattern should be embellished with messages that explain all disallowed actions (e.g., through UIP 24: EXPLAINING MESSAGES). Furthermore, since the focus of attention in most programs gravitates towards the center of the screen (e.g., by UIP 26: WORK AREA TOWARDS THE CENTER), all sacred elements should be positioned towards the screen edges (i.e., with UIP 27: FUNCTION SELECTION TOWARDS THE EDGES).

UIP 14: SAVING THE PROGRAM STATE

Context

If the user is allowed to tailor the desktop (e.g., through UIP 7: PERSONALIZABLE SCREEN LAYOUT), then the screen layout must be saved so that the user does not have to repeat the process. Furthermore, in order to complete UIP 4: DISTINGUISHING FEATURES and UIP 6: EASY TRANSITION BETWEEN PROGRAMS, the state of a program must be stored prior to deactivation.

Problem

Once a user has tailored the screen layout, the layout information must be saved either manually or automatically so that the process does not have to be repeated.

Discussion

No matter how flexible a system is with respect to tailorability and transitions between applications, this flexibility will be of little or no use unless program–state information is not lost as a result of the transitions or as a result of other physical interventions (such as a system shutdown).

Therefore

Before transferring from one program to another, the state of the program that is being abandoned must be saved. Then the target program should be entered in the same state from which it was abandoned.

Embellishments

Currently none.

UIP 15: DATA SELECTION

Context

If a project addresses the need for moving (a) data from one program to another or (b) data within a single program (i.e., through UIP 5: TRANSFER OF DATA), then the user must be provided with standard methods for selecting blocks of data of varying size.

Problem

If a system is laborious with respect to data selection, then the system will be nearly useless no matter how efficient and capable it is in processing data.

Discussion

The purpose of a computer is to accomplish some task. This usually involves manipulating data, which implies that somehow the data must be specified by someone or something. Typically, the user is the one who specifies the data upon which operations will be performed. In fact, data specification is probably one of the most important tasks a user possesses during the cooperative efforts of a user and a machine. This suggests that the user–interface designer must carefully consider how data are selected.

Most often there will be levels of selection. For example, a text document consists of paragraphs, which consists of sentences, which consists of words, which consists of characters. A drawing might consist of objects which may be either overlapping or nonoverlapping.

Therefore

For each program, the smallest selectable object must be defined; and the user should be allowed to select any block made up of these objects. In a mouse–oriented system, the user should be able to position the mouse at the beginning (or end) of the selection and move it to the end (or beginning) while being in select mode. If natural levels of selection have been identified, methods should be provided that allow data blocks at the different levels to be selected just as easily as it is to select the smallest selectable object. Several Select commands may be provided; for example, a Select–All command could select an entire document, but should be supplemented by a Select–Sentence command as well as a Select–Word command.

Embellishments

In order to complete UIP 15: DATA SELECTION, the fact that some objects do not fit on one screen and therefore rely on windows that can be scrolled must be addressed (e.g., through UIP 28: AUTOSCROLL). Inclusion of UIP 18: LEVELS OF VIEW and UIP 30: ABSTRACTION WITHOUT RESTRICTIONS allow data selection to be independent of the level of detail at which the data are displayed. Finally, a novice user will start with the most primitive selection methods available, but will quickly want to go about the process in a more efficient manner (e.g., through UIP 29: SHORTCUTS FOR THE EXPERT).

UIP 16: DATA INSERTION

Context

If the user has been given the capability to transfer data (i.e., through UIP 5: TRANSFER OF DATA), then consideration must be extended towards how and where to insert the data.

Problem

Depending on the task and the nature of the data, the point of insertion may not appear to be well–defined.

Embellishments

At the surface level, this pattern may look trivial; and in some cases it is. In a word–processing program, the point of insertion is indicated by the position of the caret. If a chunk of text is being inserted, then the leftmost character in the chunk is inserted at the position of the caret.

However, the point of insertion is not obvious when using programs such as MacPaint and MacDraw. MacPaint always inserts data in the middle of the active window, whereas MacDraw places the inserted data in the active window approximately where the user most recently clicked the mouse. In both cases, the user may reposition the inserted data without explicitly selecting it; and the insertion routines leave the data selected.

Therefore

Unless the data–insertion point can be specified in a manner that leaves no room for error, the user must be allowed to reposition the data immediately after the block has been displayed on the screen.

Embellishments

In order to complete this pattern, the user must be given some way to move data (typically graphical data) from one place to another. Therefore, a pattern must be developed to address the need for and the nature of draggable regions. Finally, the user should be allowed to insert data independent of the level of view (e.g., through UIP 18: LEVELS OF VIEW).

UIP 17: SHAREABLE SCRAPBOOK

Context

For any system that allows the transfer of data between programs (i.e., file import/file export through UIP 5: TRANSFER OF DATA), there must exist some common method to store and access the data that are being transferred. Therefore, a utility should be provided which has the sole function of moving data from one context to another.

Problem

Users will refrain from merging data produced by different programs unless the effort required is minimal.

Discussion

Experience has demonstrated that one of the most confusing aspects of using word processors, database packages, spreadsheets, and drawing programs arises when a user is asked to merge data produced by the different programs. A possible reason for this is that users are made aware of such capabilities late in the learning process; thus, they may maintain the misconception that such capabilities do not exist or they believe that they are difficult to use.

Therefore

On all systems, a utility should be provided which serves as an intermediate–storage area; ideally, this would allow the user to store data using any program and also to retrieve data using any program. The user must be made aware of this utility early on in the learning process; in this way, users will understand that merging data of different kinds can be the rule, not the exception.

UIP 18: LEVELS OF VIEW

Context

After the elements of the activity center have been defined (in UIP 2: ACTIVITY CENTER), and after a decision has been made regarding how the user may tailor the interface (in UIP 7: PERSONALIZABLE SCREEN LAYOUT), a decision must be made regarding the different ways the user should be allowed to view the data. In most domains, the ability to examine data at different levels of abstraction is crucial; in addition, there is a need to zoom in on and zoom out from particular areas (see Figure 17).

Problem

Users have a need to examine data displayed at different levels of detail.

Discussion

The hypothesis behind this pattern is that people have a need to survey the world around them at different levels of abstraction. Alexander et al. (1977) argues that people have a fundamental need to climb up to high places in order to look down on their surroundings. Similar needs are present in the domain of user interfaces. One reason relates to the fact that the available work space (the screen) is very limited in size. Consequently, when users write a document or draw a picture, they only have a partial view of the data. The screen acts as a window through which subparts of the data can be viewed. Users often find themselves editing data without being able to view the data in a larger context.

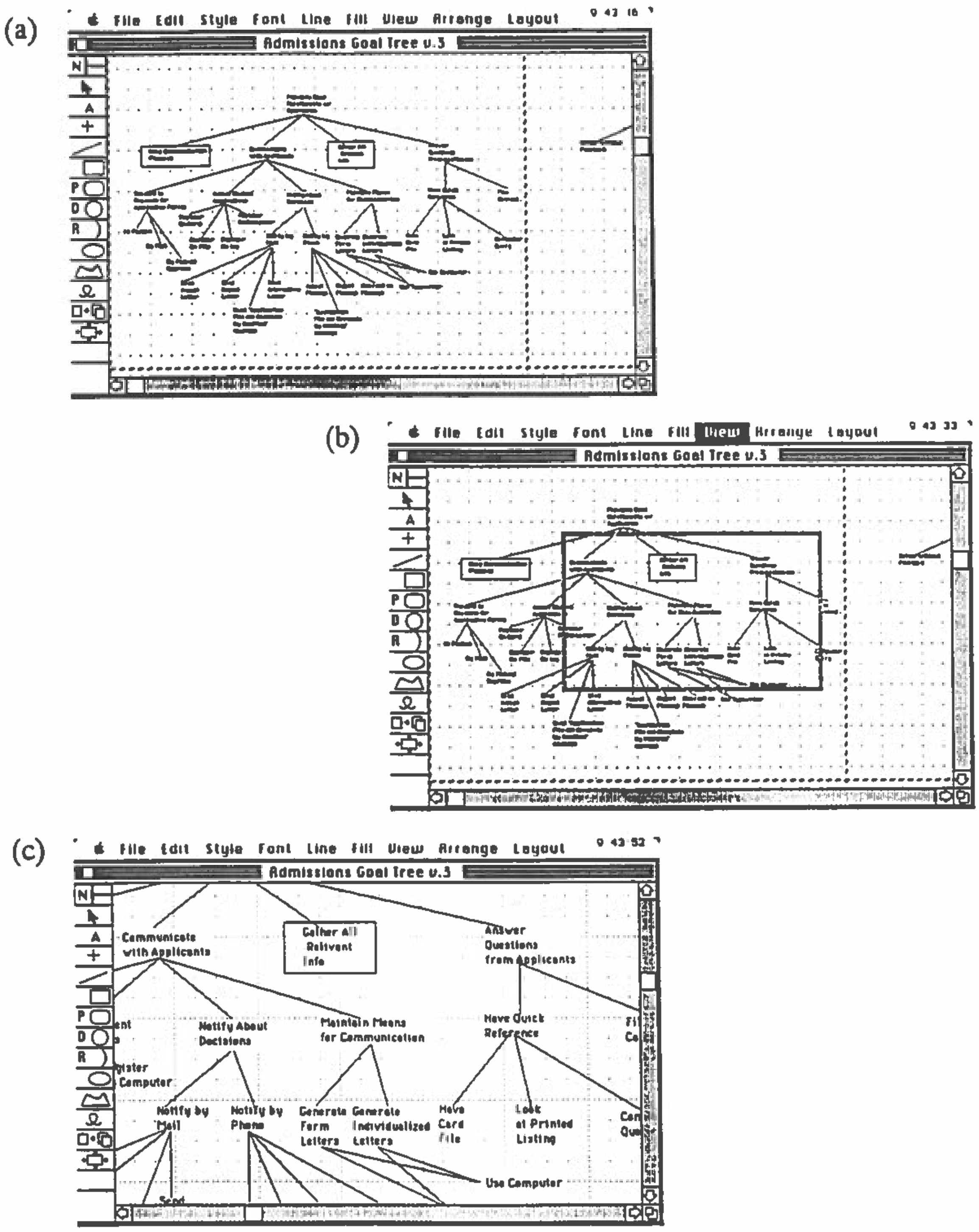

(a)
File Edit Style Font Line Fill View Arrange Layout
Admissions Goal Tree v.3
(b)
File Edit Style Font Line Fill View Arrange Layout
Admissions Goal Tree v.3
(c)
File Edit Style Font Line Fill View Arrange Layout
Admissions Goal Tree v.3
Communicate with Applicants
Gather All Relevant Info
Answer Questions from Applicants
Notify About Decisions
Maintain Means for Communication
Have Quick Reference
Computer
Notify by Mail
Notify by Phone
Generate Form Letters
Generate Individualized Letters
Have Card File
Look at Printed Listing
Use Computer
Send

FIGURE 17. Viewing Data at Different Levels of Abstraction (*a* portrays a tree being viewed at a high level of abstraction; *b* portrays a selected portion of the tree; and *c* portrays the selected portion viewed at a lower level of abstraction)

A secondary reason concerns the fact that it is often desirable to hide detailed information until the user specifically requests it. In most cases, the need is sufficed with a high–level view, and the more detailed information is typically of a lesser concern.

There are several simple ways the designer can address the needs of the users in this case. In many programs there is a *Reduce/Enlarge* command. In the ideal case, the user may select the data and then reduce or enlarge it to a specified percentage. There is also the notion of node expansion; in fact, an example of this is Folders on the Macintosh desktop. Since it allows the user to select the level of detail with respect to file information, the View menu on the Macintosh desktop also demonstrates an implementation of UIP 18: LEVELS OF VIEW.

As long as user–interface designers are aware of the existence of UIP 18: LEVELS OF VIEW (both as a result of fundamental human instincts and as a result of hardware limitations), the situations in which it should be embedded within the design will be nearly unmistakable.

<u>Therefore</u>

The user should always be allowed to view data from different perspectives; this may be accomplished by including methods for physically reducing and enlarging the size of the data. In addition, the user must be able determine the level of detail at which the data should be displayed by using the idea of node expansion. (That is, when a node is expanded, more detailed information becomes available.)

Embellishments

In addition to allowing the user to choose the level of abstraction at which data are viewed, the user should also be provided with the choice to view data in different forms. For example, histograms as well as tabular views, and textual as well as icon–levels. Thus, a pattern needs to be developed to address this issue. Finally, the ability to edit data should be independent of the level of abstraction––as much as is practically possible (e.g., through UIP 30: ABSTRACTION WITHOUT RESTRICTIONS).

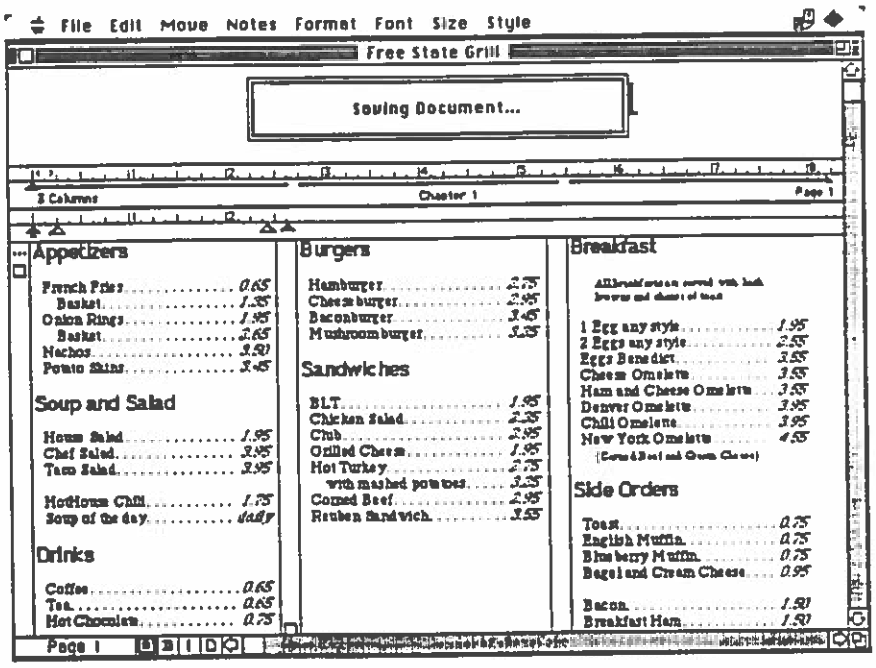
UIP 19: IMMEDIATE FEEDBACK

Context

For every command or option that is carried out or selected by the user, there must be *some* indication that the action has been recognized by the system. This *something* must also indicate to the user which command was actually carried out. The user may think that Command A was selected, whereas the user selected Command B by mistake (see Figure 18).

Problem

Unless the user is given unambiguous, visually representative feedback when (a) the computer responds to a command initiated by the user and (b) when a function has been selected, the user will feel insecure about the activities which are taking place.



Appetizers

French Fries	0.65
Basket	1.35
Onion Rings	1.95
Basket	2.65
Nachos	3.50
Potato Skins	3.45

Soup and Salad

House Salad	1.95
Chef Salad	3.95
Taco Salad	3.95
HotHouse Chili	1.75
Soup of the day	daily

Drinks

Coffee	0.65
Tea	0.65
Hot Chocolate	0.75

Burgers

Hamburger	2.75
Cheeseburger	2.95
Baconburger	3.45
Mushroomburger	3.35

Sandwiches

BLT	1.95
Chicken Salad	2.35
Club	2.95
Grilled Cheese	1.95
Hot Turkey	2.75
with mashed potatoes	3.25
Corned Beef	2.95
Reuben Sandwich	3.55

Breakfast

[illegible]

1 Egg any style	1.95
2 Eggs any style	2.55
Eggs Benedict	3.55
Cheese Omelette	3.55
Ham and Cheese Omelette	3.55
Denver Omelette	3.95
Chili Omelette	3.95
New York Omelette	4.55
(Corned Beef and Cream Cheese)	

Side Orders

Toast	0.75
English Muffin	0.75
Blueberry Muffin	0.75
Bagel and Cream Cheese	0.95
Bacon	1.50
Breakfast Ham	1.50

FIGURE 18. FullWrite During a Save Operation

Discussion

This pattern is a first attempt towards specifying how users can feel confident that they "did the right thing." Experience has demonstrated that many interfaces are unacceptable in this respect. Five short examples illustrate this point:

1. The Open and Save commands are some of the most important commands in most programs. One of the first lessons a computer novice is given is to "remember to save your data!" One of the most common remarks about the Save command is typically that "I selected the Save command, but nothing happened; have I lost everything now?"

2. During basic text editing several problems arise. One that is particularly interesting is a problem due to the Copy command. On the Macintosh, when a block of text has been specified and the Copy command is selected there is no visible change. A significant number of novices repeatedly select the Copy command expecting something to happen which can be seen.

3. An obvious example of the lack of feedback is the manner in which the Grid command is handled in MacPaint. An informal, 3–year study that has been carried out in a user–interface class taught by Dr. Sally Douglas at the University of Oregon has indicated that novices are utterly confused by the fact that a grid is not displayed when the Grid option is selected. In fact, because a grid is not displayed the first time, most users select the Grid option yet another time. In most cases, the users/subjects did not understand the problem until the experimenters interrupted and explained the situation.

4. The problem with the Grid commands indicates that check–mark commands (such as the Grid command in MacPaint) and toggle commands (such as Turn Grid On versus Turn Grid Off in MacDraw) are difficult to grasp for the first–time user. The users who had problems with the Grid command obviously did not notice the checkmarks that appeared and disappeared interchangeably. And, when users select the Turn Grid On command, no visible feedback is given; the problem is that the change from On to Off (or vice versa) cannot be seen without pulling down the menu.

5. When the spelling checker is activated in Microsoft Works, the dictionary must be loaded. Since the load operation requires a certain period of time, the interface designers chose to display a spinning beach ball in order to

indicate that a lengthy operation is in progress. Most students become very worried when the beach ball appears.

Once the user–interface designer is made aware of the lack–of–feedback problem, negligible effort is required to eliminate most of these problems. However, there obviously is no simple solution to all of these problems. In particular, the Copy problem is difficult to solve in a manner that will satisfy both novice and expert users. Each problem may be considered separately.

The Save problem is handled nicely in FullWrite. In fact, all actions that require a certain length of time should notify the user that ongoing processing is occurring. FullWrite displays a message while commands are being carried out; it is simple and effective.

One way that the Copy command can provide feedback is to display a short message after Copy has been selected. Such a message would undoubtedly be a nuisance to the more experienced users. This might indicate that several levels of feedback should be available in a program.

The solution to the third problem is obvious. The grid could be drawn (as in MacDraw). A related function is the Snap On/Off command; this command, however, is far more difficult with respect to visual feedback.

Checking–off menu items and changing menu items has been integrated into most Macintosh applications. These methods are excellent for proficient users, but experience has shown that novice users do not discover the many subtle changes that occur. It becomes even more confusing when checked menu items, toggled menu items, and regular menu items are grouped together.

Finally, the spinning beach ball and other such puzzling symbols should be replaced by a message. Whenever possible, words should be used instead of obscure icons.

Therefore

The solution is to provide the user with ample feedback (a) by displaying messages which indicate which actions are being carried out by the computer, (b) by displaying messages confirming that the system has registered a command selection, and (c) by including auditory signals (as described in Embellishments, the next section).

Embellishments

There is no point in making users feel confident about their actions by providing feedback unless the user is able to reverse the actions (i.e., by performing UIP 20: REVERSIBLE ACTIONS). UIP 19: IMMEDIATE FEEDBACK should be embellished by considering UIP 10: PALETTES and UIP 24: EXPLAINING MESSAGES. Another possible solution involves creating a UIP that addresses the feedback issues which arise as a consequence of direct–manipulation interfaces. And, finally, UIPs may be created to address auditory feedback–related issues (e.g., as discussed by Deatherage, 1972).

UIP 20: REVERSIBLE ACTIONS

Context

It is a fact of life that people make mistakes. In the domain of computers, both novices and experts make mistakes and spend considerable time correcting the same. Once a user discovers that a mistake has been made, it is part of the program's responsibility to provide methods for undoing it. This pattern should be considered within the same context as UIP 8: LIMITED NUMBER OF COMMANDS and UIP 19: IMMEDIATE FEEDBACK (see Figure 19).

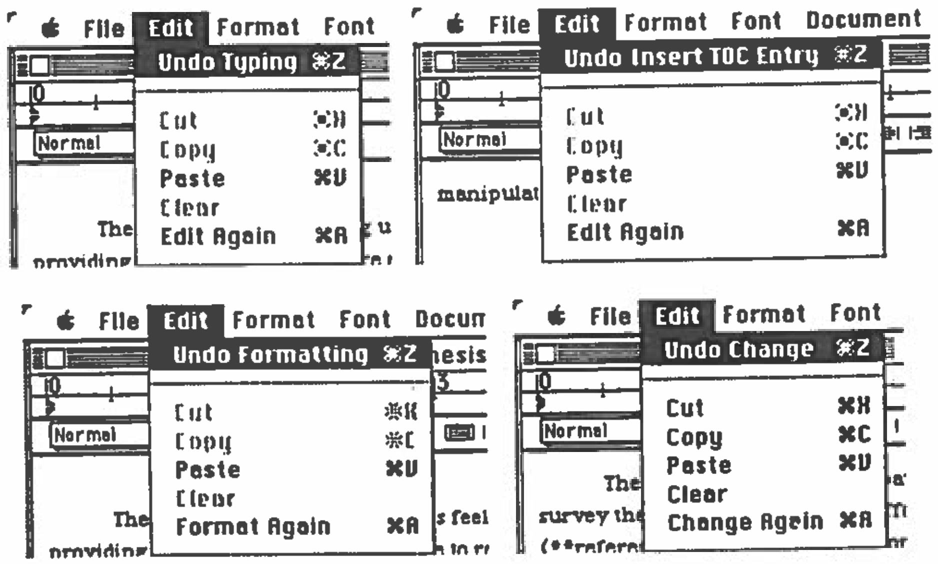


FIGURE 19. Examples of Actions That Can Be Undone

Problem

It is impossible to build software systems that can anticipate all possible errors that can be made by the user.

Discussion

Anticipating and undoing errors are some of the most difficult problems in program design as well as in user–interface design. Suppose that a document consisting of four paragraphs is being edited. The order of the paragraphs is

Paragraph A–Paragraph B–Paragraph C–Paragraph D. Further imagine that the document is to be shortened to Paragraph A–Paragraph B–Paragraph D; but, by mistake, the result becomes Paragraph A–Paragraph D. Obviously, the system cannot know that a mistake has been made. Should Paragraph B have to be retyped? A good system would provide the user with a more attractive option: to undo the command that deleted Paragraph B. Consequently, the system must, in effect, assume that the user is constantly making mistakes.

Unfortunately, there are some practical aspects related to the Undo option that are troublesome. Typically, only the most recent action can be undone. If Undo is selected twice, the first undo will be undone. Hence, in the preceding Paragraph example, if the user first deleted Paragraph B and then deleted Paragraph C, Paragraph B would be completely lost. Experience has shown that mistakes are not discovered right away; thus, the Undo command is useless in many situations. One possible solution is to provide a variation of the regular Undo function; this variation may be called *Undo From Stack* with its name suggesting that commands can be undone in a last–in/first–out fashion. Obviously, Undo From Stack would be very memory intensive. In practice, the size of the stack must be limited; when it is full, the least–recent action on the stack would need to be deleted in order to make room for a new action.

There will exist commands that cannot be reversed. Typically, such commands include Erase Disk (or Format), Empty Trash, Copy one disk onto another, and in some cases Save. When commands such as these are selected, the user must be informed that an irreversible action is about to take place.

Therefore

In every program, the option to undo the most recent action (Undo) and the option to undo several actions (Undo from Stack) should be included. If a command will lead to an irreversible state change (for practical reasons), the user should be warned and allowed to cancel the command.

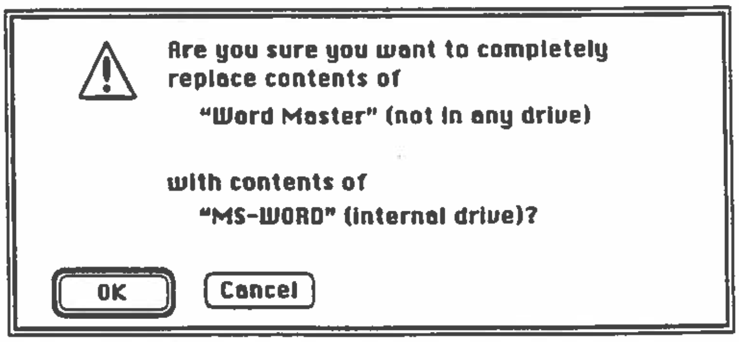
Embellishments

Because there are several actions that are irreversible, this pattern should be embellished with UIP 21: WARNING THE USER.

UIP 21: WARNING THE USER

Context

On most computer systems, there exists a set of *dangerous* commands. The user needs to be protected against the consequences of these commands. In general, the dangerous commands are those that are irreversible––as previously noted in UIP 20: REVERSIBLE ACTIONS (see Figure 20).


Are you sure you want to completely replace contents of
"Word Master" (not in any drive)

with contents of
"MS-WORD" (internal drive)?

FIGURE 20. A Warning Message With the Option to Cancel

Problem

Users will become alienated from the idea that a computer is a friendly and useful tool if mistakes with serious consequences are experienced during the early periods of learning.

Discussion

Most computer users would probably agree that mistakes leading to loss of information that is impossible or difficult to recreate are among the most serious mistakes that can occur within the domain of computer usage. Both novices and experts make such mistakes. For example, a file may inadvertently be erased. It seems that the only way such errors can be prevented from occurring is to provide users with messages that warn about the consequences of their actions. One problem is that such warnings are usually followed by a confirm or disconfirm action by the user, and periodically it happens that users select OK (confirm) instead of Cancel (disconfirm). Such *slips* are impossible for the machine to detect.

Seemingly unimportant and subtle aspects of a system can often result in considerable damage. The following true story will serve to show that users must be protected from themselves:

> Person X had been using Microsoft Word 4.0 for a period of 2 days. During these 2 days, he had been editing several files that were eventually going to be transferred to a different type of machine. Therefore, whenever he saved the data, he saved it as Text Only. He even set the default file format to Text Only. A few days later, X had to use the word processor again, but now different fonts, sizes, tabs, indentations, and so forth were used. When X saved his finished document, all the layout information was lost.

In this case, it would have been relatively simple to implement a routine that would have prevented the data from being lost:

> Program the machine to check for formatting information. If formatting information exists, warn the user that the format information will be lost if the Save command is carried out. In addition, inform the user of the necessary steps that must be taken to save all format information.

Therefore

The user must be warned if an action that will lead to loss of information (of any type) is about to be carried out. The user also should be allowed to cancel the command, and/or be informed regarding which steps to take in order to save the information.

Embellishments

UIP 21: WARNING THE USER requires a UIP which addresses default actions and states (perhaps a UIP entitled *Default Actions and States*, which is not included within this thesis document). For example, when a program asks whether or not the user wishes to exit from the program without saving the work that has been performed, it be easier to answer *no* than *yes*.

UIP 22: LIMITING THE EFFECTS OF THE DIALOG BOTTLENECK

Context

There are several bottlenecks which hamper the dialog (i.e., the flow of information) between the user and the computer.

Problem

A computer will never be perceived as a useful tool as long as users have the feeling that they are doing work that the computer could have done.

Discussion

The name of this pattern (Limiting the Effects of the Dialog Bottleneck) may be somewhat misleading. The actual point is that a system should minimize the work that must be carried out by the user. This goal can be accomplished in several ways, two of which are listed:

1. The interaction between a novice user and a system can be simplified by relying on command recognition, rather than on command recall.

2. The system should supply default values whenever possible. For example, a statistics program might use $p = .05$ as a default value when the user is requested to supply the significance level for a t test; as an additional extension, if the user enters a different value, that value could be used as the default for the next request.

Therefore

Instead of requiring the user to carry out the trivial tasks, the computer could be designed to execute these whenever programmatically possible. Ideally, domain knowledge would be utilized to supply default values, but then would employ the user's previous inputs as bases for default values. This does not imply that it is necessary to incorporate advanced Artificial Intelligence technology into every program; it is often the small and simple things that have the most greatest effect.

Embellishments

UIP 23: USER IN CONTROL should be included, and UIPs should be created which address issues related to command recognition rather than to command recall. For example, command recognition could be promoted by the user interface. Currently, the use of palettes and pull–down menus are popular, and user–interface patterns for these have been developed; but perhaps there are different and better methods. A final recommendation would be to restrict UIP 22: LIMITING THE EFFECTS OF THE DIALOG BOTTLENECK so that the computer does not control the human/computer interactional dialog that takes place.

UIP 23: USER IN CONTROL

Context

UIP 22: LIMITING THE EFFECTS OF THE DIALOG BOTTLENECK addressed the requirement that a system should minimize the work that must be carried out by the user. However, since there is a danger in taking this recommendation too far, UIP 22: LIMITING THE EFFECTS OF THE DIALOG BOTTLENECK must be restricted in some way.

Problem

Users should perceive the computer as a tool. Users should not perceive themselves as tools being used or manipulated by the computer.

Discussion

In a series of informal experiments carried out over the last 3 years by students in an introductory user–interface class taught by Dr. Sally Douglas at the University of Oregon, several problems with the MacPaint and MacDraw user interfaces have been recognized. One of the most prevalent problems can be attributed to mode changes that are carried out by the programs as side effects of the user's actions. More specifically, in MacDraw, after having selected and used a palette function (e.g., selecting the rectangle function and drawing a rectangle), the palette selection resets to a default function. Consequently, if the user wants to draw several rectangles in succession, the rectangle function must be repeatedly selected for each rectangle. In the informal studies previously referenced, this inconvenience was found to be one of the most serious problems with the MacDraw user interface.

A user expects to be in control over mode changes. In the MacDraw scenario, the problem concerns the fact that each function in the palette causes the program to enter a different mode, and the user selects a particular mode (or function) with a deliberate action. The difficulty is that a mode change occurs as a side effect of the user's actions.

It could be argued that a mode change should never occur unless it is deliberately initiated by the user. That is, in general, the system should provide default selections and initiate mode changes if, and only if, the following conditions exist:

1. If the default choice provided by the computer is not what the user desires, then the user should not need to perform more work to change the choice than if the default choice had not been provided at all.

2. If the mode change initiated by the computer results in a different mode than that desired by the user, then the user should not need to perform more work than if the mode change had not been carried out at all.

Therefore

All actions that lead to a change in a system's state should be deliberately initiated by the user. The only features which should be considered are those in which the circumstances involve the system supplying default values or initiating mode changes without increasing the user's work load.

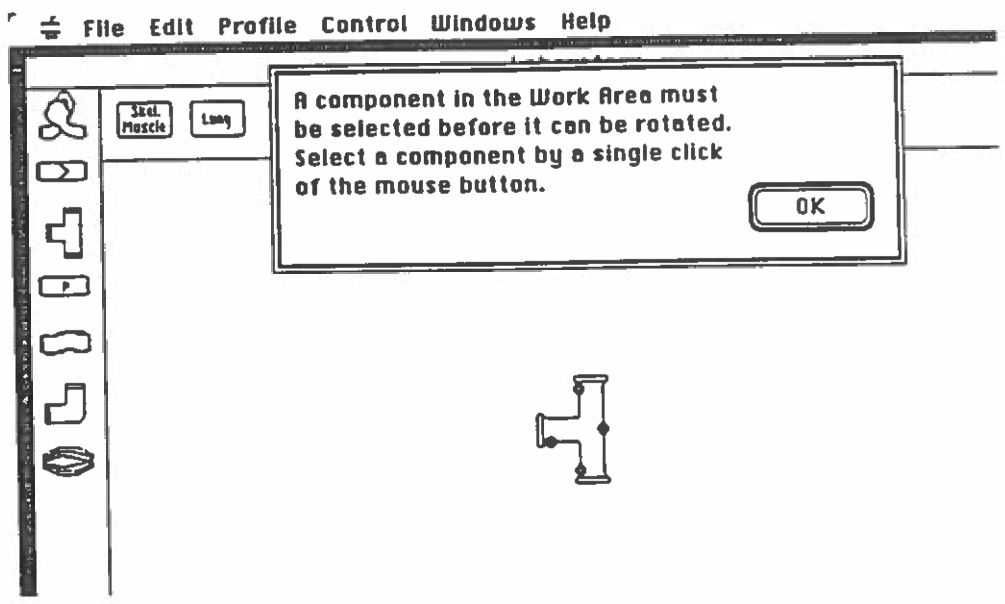
Embellishments

Currently none.

UIP 24: EXPLAINING MESSAGES

Context

Some of the computer operations which occur are so time–consuming that the user may be led to believe that something is wrong. In addition, there are occasions when the system must prevent the user from carrying out certain actions (which relates to UIP 13: PROTECTED ELEMENTS); there are, as well, some functions which might not be applicable in the system's current state (see Figure 21).



A component in the Work Area must be selected before it can be rotated. Select a component by a single click of the mouse button.

FIGURE 21. A Message Explaining Both a Problem and How to Fix It

Problem

Users need to feel in total control over a computer. If the user ever perceives the computer to have its own goals, then the user never will be able to feel comfortable with it; that is, the user will perceive that the computer is no longer a tool, but a competitor.

Discussion

The following types of comments from computer novices are not atypical: "It has been doing it for a while now; what's taking so long?," "I don't understand why it won't let me do that," and "This is what I selected, but the computer didn't do anything." One of the most important aspects of user–interface design is to create an interface that does not lead to such utterances. Several steps may address this situation:

1. During a lengthy operation, the computer should display a message to inform the user of the operations that are taking place. For example, this method would apply when data are being loaded from or saved to a disk. It should be ensured that the message is displayed long enough so that the user is able to read it, even though the operation itself might require less time.

2. During a lengthy operation, the user should be informed regarding how much work has been completed and how much work remains undone. If the user realizes that the operation will require too much time, then the option to cancel the command should be available (see Figure 22).

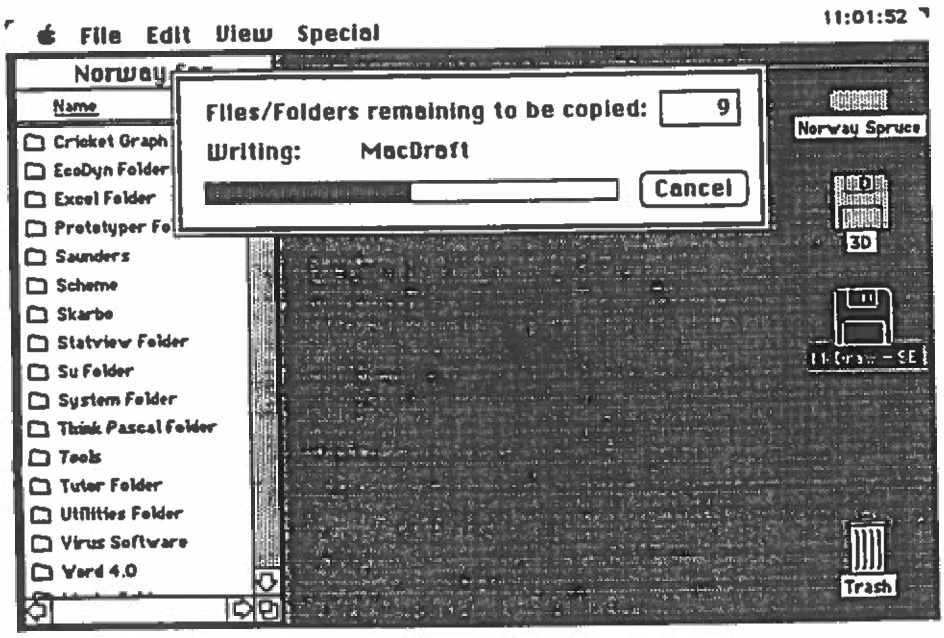

Files/Folders remaining to be copied: 9
Writing: MacDraft

FIGURE 22. A Message Indicating Amount of Work Remaining

3. The Macintosh user–interface guidelines (ACI, 1987) specify that menus, menu items, buttons, radio boxes, and check boxes that do not apply in a given state of the system must be disabled (i.e., dimmed). This guideline does more harm than good. The problem is that the user is seldom able to infer the reason why a command has been disabled; moreover, there are no indications regarding which steps need to taken in order to enable a certain command. The problem could be reduced by changing the semantics of the term *disabled.* Selecting a disabled command should result in the display of a message that describes the steps required to reenable the command.

Therefore

The user should always be provided with messages that will help him or her understand the activities of a program. If the program needs a few seconds to accomplish *X*, the computer should simply state this fact so that the user knows that *X* is, indeed, being carried out. Furthermore, if a command does not apply within a certain program state, the users should not need to infer the steps required to apply the command; the users should be simply told what must be done.

Embellishments

In order to make UIP 24: EXPLAINING MESSAGES work, all programs within the programming community must use the same kinds of messages. This is related to UIP 9: STANDARDIZED METHODS. UIP 24: EXPLAINING MESSAGES also creates a problem related to the differences between novice and expert users. The expert may find the messages annoying; whereas the novice may find them helpful, even crucial, for understanding the program. Therefore, this pattern should be embellished with the ideas contained in UIP 18: LEVELS OF VIEW (i.e., the messages might vary in detail depending on the user), and those within UIP 29: SHORTCUTS FOR THE EXPERT (which could eliminate the messages altogether, as implied by UIP 18: LEVELS OF VIEW). Finally, any lengthy action should be interruptible by the user, in which case the original program state would be restored (e.g., through UIP 20: REVERSIBLE ACTIONS).

UIP 25: ELEMENT BOUNDARIES

Context

Once it has been decided which elements to include in an interface (i.e., given UIP 3: STANDARD SET OF ELEMENTS, UIP 10: PALETTES, UIP 11: PULL–DOWN MENUS, and UIP 12: MANIPULABLE WINDOWS), the problem which stems from displaying a large number of elements simultaneously on the screen must be addressed. The boundaries between the elements can easily become blurred unless careful attention is given to the physical boundaries between the elements (see Figure 23).

Problem

Each element of a user interface has unique responsibilities. These responsibilities cannot be accomplished unless each and every interface element is visually separable from all the other elements.

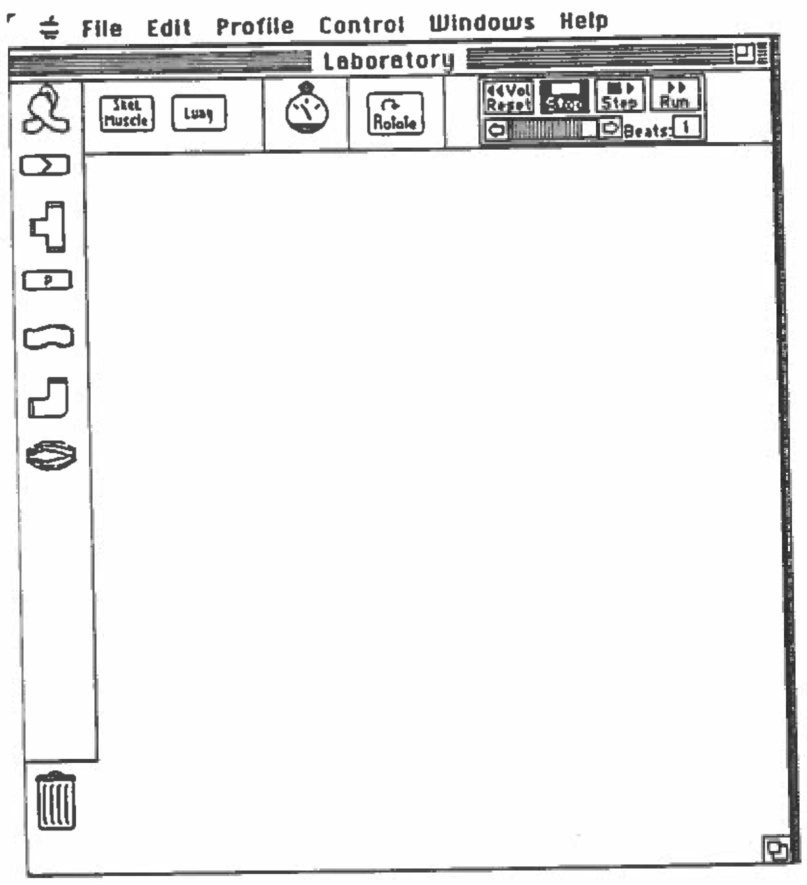


FIGURE 23. An Illustration of the Simplicity Created in an Interface When Effective Element Boundaries (Simple Lines) are Included

Discussion

By comparing Figures 23 and 24, one may note that Figure 24 illustrates the elimination of some of the most important boundaries which are contained in Figure 23. In Figure 24, there no longer is a well–defined notion of the work area and function–selection area. It is unclear exactly where the title bar is, and the exact height of the menu bar is far from obvious.

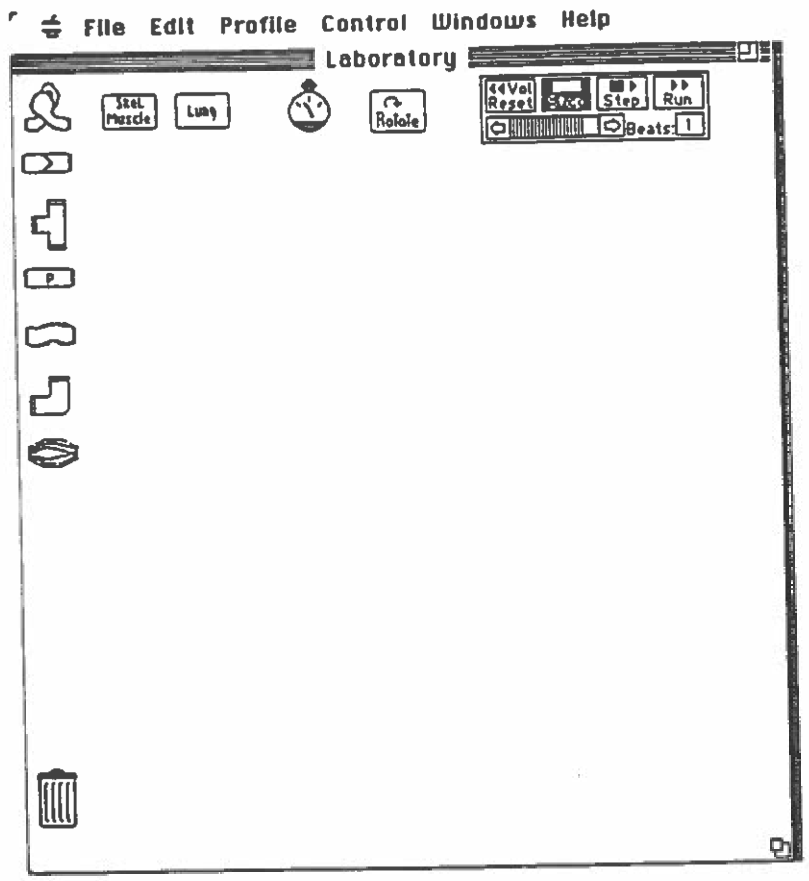


FIGURE 24. An Illustration of the Confusion Created in an Interface When Element Boundaries are Excluded

<u>Therefore</u>

The various user–interface elements should be separated by including physical boundaries. Quite often either a simple line between the elements, or a portrayal which utilizes different background colors/patterns, is enough to separate the elements.

Embellishments

Currently none.

UIP 26: WORK AREA TOWARDS THE CENTER

Context

Once it has been decided how to present the various options and functions to the user (e.g., through UIP 10: PALETTES, UIP 11: PULL-DOWN MENUS, and UIP 12: MANIPULABLE WINDOWS), and elements that need to be protected have been identified (e.g., through UIP 13: PROTECTED ELEMENTS), a decision must be made regarding where and how the work area should be arranged; after all, the work area is where the various functions will be applied.

Problem

Given a blank sheet of paper, people prefer to scribble towards the middle of the sheet and stay away from the edges.

Discussion

UIP 26: WORK AREA TOWARDS THE CENTER involves a very interesting concept despite the fact that it may seem obvious. It should be noted that most Macintosh interfaces are designed so that the user works in an area directed towards the center of the screen. Menus and palettes, on the other hand, gravitate towards the edges of the screen. There are practical reasons for this arrangement. For example, if a palette were to be placed in the middle of the screen, the uninterrupted work area would be significantly reduced. UIP 26: WORK AREA TOWARDS THE CENTER also recognizes the more fundamental human instincts. If a typical office desk, for example, is considered, one may note that all the functionality elements (such as the telephone, the stapler, the pencil sharpener, the pen–and–pencil holders, and so on) are located towards the edges of the desk; the work space is in the middle of all this. An informal experiment that provided interesting results was conducted during the course of this study. The subjects (15 peers) were given a blank sheet of paper (measuring 8.0 x 11.5 inches) and a pencil; they were then asked to "draw a circle less than 2.0 inches in diameter." Of the 15 subjects, 14 people drew the circle right in the middle of the paper and only 1 person drew the circle close to the edges of the paper.

The preceding observations obviously are not provided in order to prove any theories. They are presented because they are good indications that UIP 26: WORK AREA TOWARDS THE CENTER should be maintained, both for the practical as well as the human–factor reasons.

Therefore

Users should be allowed to carry out their work in an area that tends towards the center of the screen. In addition, users should always be permitted to move the data inside a window so that the work can be carried out in the center of the window.

Embellishments

UIP 26: WORK AREA TOWARDS THE CENTER depends on UIP 27: FUNCTION SELECTION TOWARDS THE EDGES.

UIP 27: FUNCTION SELECTION TOWARDS THE EDGES

Context

The function selectors should be located towards the edges of the screen in order to support UIP 26: WORK AREA TOWARDS THE CENTER.

Problem

People tend to organize their work space so that the tools they need are conveniently located in a manner such that the tools do not obstruct the work that is to be accomplished.

Discussion

The discussion for UIP 27: FUNCTION SELECTION TOWARDS THE EDGES is contained within the preceding discussion for UIP 26: WORK AREA TOWARDS THE CENTER.

Therefore

Palettes and menu bars should be located towards the edges of the screen. If it becomes necessary to relate functionality to individual windows, then the selectors should be placed towards the edges of the windows.

Embellishments

UIP 27: FUNCTION SELECTION TOWARDS THE EDGES depends on UIP 26: WORK AREA TOWARDS THE CENTER.

UIP 28: AUTOSCROLL

Context

When the means for selecting data (e.g., through UIP 15: DATA SELECTION) have been included in a system, then the methods by which large blocks of data can be easily selected must also be provided.

Problem

The user should be able to select any subset of some data block just as easily, and in the same way, as the atomic (and, thus, smaller) parts of data are selected.

Discussion

The pattern UIP 28: AUTOSCROLL is related to UIP 18: LEVELS OF VIEW. By activating UIP 18: LEVELS OF VIEW, the data could be reduced to an appropriate size prior to selection. UIP 28: AUTOSCROLL is a pattern which contends that it also should be possible to select any subset of a large data set without changing the level of view.

A rather annoying aspect of MacPaint seems appropriate to this discussion. If a small part of the work space needs to be selected, it is relatively simple to utilize the Lasso function. However, if some part of the picture does not fit into the window, but has to be selected, then additional steps are required. Such additional steps are great irritation factors for both novice and expert users. Experience has demonstrated that users learning MacPaint find the data–selection aspects of the interface extremely troublesome.

The latest version of MacPaint (version 2.0) handles data selection excellently. If the selection is larger than the window, then the window contents automatically move in a direction indicated by the movement of the mouse.

Therefore

A program must assist the user during the data–selection process. It always should be possible to select chunks of data in one step––regardless of size. In a word–processing program, for example, selecting an entire document should not be conceptually different from selecting one letter. UIP 28: AUTOSCROLL is adequately satisfied by having the system automatically adjust the window through which the document is viewed (through the technique labeled *autoscroll*).

Embellishments

Currently none.

UIP 29: SHORTCUTS FOR THE EXPERT

Context

Once decisions have been rendered regarding the types of functionality to include in a program and the plan for how the user is to interact with the program (i.e., UIP 10: PALETTES, UIP 11: PULL–DOWN MENUS, UIP 15: DATA SELECTION, and UIP 16: DATA INSERTION are established), then the fact that users may be categorized into several proficiency groups––ranging from novices to experts––must be considered.

Problem

Novice and expert users possess differing, and often opposing, needs with respect to user–interface design.

Discussion

Some inherent difficulties in user–interface design arise from the fact that the various proficiency levels of the users must be reflected in the user interface. A user–interface design is, in essence, supposed to be a model of user behavior. However, it is very difficult to design an interface which is able to satisfy the needs of all users at all proficiency levels. UIP 29: SHORTCUTS FOR THE EXPERT does not totally solve this problem; nevertheless, it has been included in order to increase designers' awareness of the issue and a few partial solutions that have been integrated into several software products are outlined.

There is one fact of life that makes it challenging to arrive at a solution to the preceding problem: Program functionality and program complexity are positively related. Therefore, how much functionality can be left out before the expert user becomes unhappy, and how much functionality can be added before the system becomes incomprehensible to the novice? A designer must incorporate steps to design easy–to–learn programs which are, simultaneously, interesting for expert users. For example, when using MacDraw, a novice user can accomplish a large number of tasks by adhering to the functions available from the palette; then, as a user becomes more accomplished, the more advanced functions located within the menus can be utilized.

The fact that functionality and simplicity are inversely related does not necessarily imply that any tradeoffs must exist. Both novice and expert users can be satisfied. There will eventually be innovative user–interface designs that will satisfy all user groups; in fact, this eventuality can be already seen in some products available at the present time. One solution is to allow the user to select the level of functionality and to prescribe the complexity of the system to adjust accordingly. For example, Microsoft Word has two functionality (and, thus, two complexity) levels (see Figure 25). Microsoft Word also allows expert users to program keyboard equivalents for all the commands that are available.

In summary, the problem can be reduced to defining which functions are appropriate at the various functionality/complexity levels. Unfortunately, this may in itself be a difficult task.

Therefore

In an attempt to increase learnability, functionality should never be left out from a program. If a function is useful (even if only to an expert user), then it should probably be included. In order to avoid an increase in learning time, the option of defining several (two or more) functionality/complexity levels should be explored. At the most basic level, only those functions which are relevant to novices would be included; but, at the most advanced level, all available functions would be available. It is most important to never eliminate functions as the complexity level increases: In other words, if *X* is the set of functions at complexity level *x*, and *Y* is the set of functions at complexity level *y*, and level *y* is more complex than level *x*, then *X* is a proper subset of *Y*.

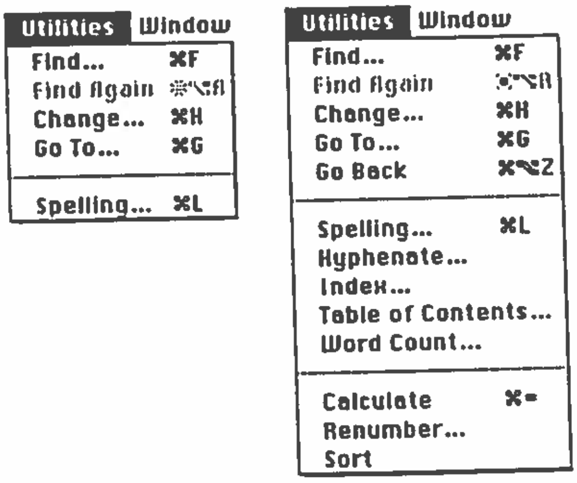


FIGURE 25. An Example of Two Different Functionality/Complexity Levels

Embellishments

UIPs should be developed to address (at a finer level of detail) the issues discussed in UIP 29: SHORTCUTS FOR THE EXPERT. Ideally, such patterns should be more explicit with respect to solutions for resolving the tension between functionality and learnability, and they should also address the issue of command keys (equivalent to menu items) in more detail. For example, any sequence of actions using the mouse should also be possible without using the mouse.

UIP 30: ABSTRACTION WITHOUT RESTRICTIONS

Context

If users are allowed to view data at several levels of abstraction (such as recommended in UIP 18: LEVELS OF VIEW), then the users also will want to edit the data independently of the display level.

Problem

One of the reasons a user may wish to view information at differing levels of abstraction is that certain aspects of the data often are better captured at a certain level of abstraction as opposed to other levels. In addition, if the user *is allowed* to modify the parameters being examined but *is not allowed* to change the level of abstraction, then UIP 18: LEVELS OF VIEW is of little or no use.

Discussion

Many existing systems have included a pattern similar to UIP 18: LEVELS OF VIEW in one form or another. Unfortunately, strange restrictions often come into effect when the level of abstraction changes. As a case in point, the Page View option in Microsoft Word displays a document according to the format in which it will appear on the printed page. With Page View, the user is able to examine margins, headers, footers, footnotes, and page numbers in addition to the data––all in normal view. The problem with older versions of Microsoft Word (versions 3.0 and earlier) was that none of these parameters could be altered while in Page View; thus, the Page View option was of little use. However, in the later update, Microsoft Word (version 4.0), the user is allowed to perform the same editing tasks in Page View as in normal view. Thus, the recommendations of UIP 18: LEVELS OF VIEW have been satisfied since changing the level of abstraction no longer, ipso facto, leads to any restrictions.

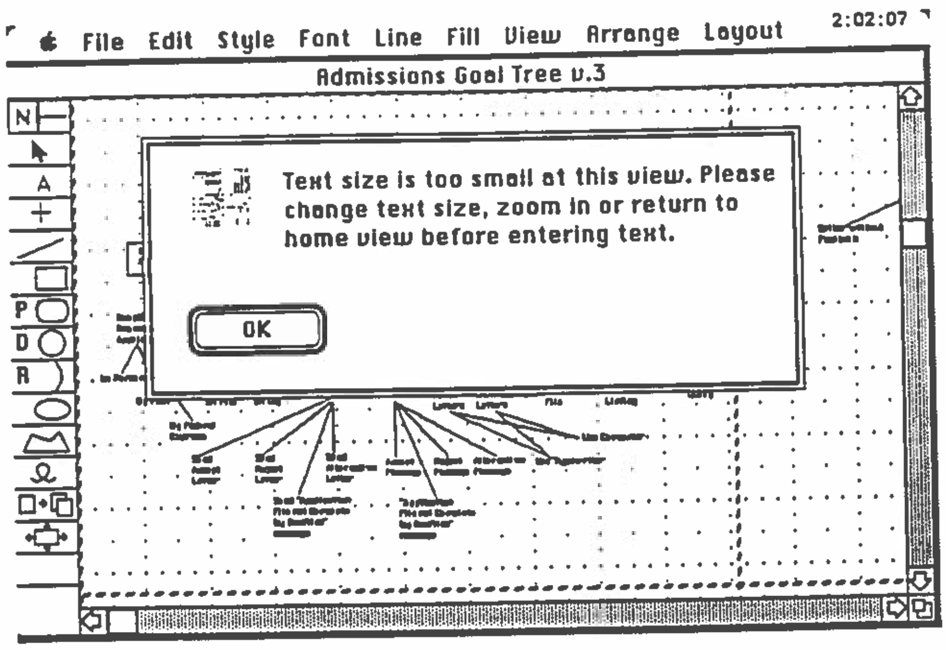
In some cases, it is natural to limit the functionality when the level of view changes. For example, Microsoft Word has a print–preview option. At this level of abstraction, it does not make sense to edit the contents of the text because the print is too small to be legible. As a general rule, a function should never be disabled when the level of view changes unless it is physically impossible to support the function.

Therefore

In most programs, there is a default view level and one or more alternative view levels. Designers should increase, rather than restrict, the availability of functions and options when the view level changes––unless the function becomes useless due to physical constraints.

Embellishments

UIP 30: ABSTRACTION WITHOUT RESTRICTIONS should be embellished with UIP 24: EXPLAINING MESSAGES given instances in which it is impossible to edit the data. Figure 26 provides such an example.



Text size is too small at this view. Please change text size, zoom in or return to home view before entering text.

FIGURE 26. A Message Clearly Explaining the Reason for Imposing a Restriction

Summary

This chapter, Chapter IV, has defined 30 user–interface patterns. Succinctly, a user–interface pattern (UIP) consists of a context in which a user–interface design problem occurs in confluence with a possible solution to the problem. The designated user–interface patterns were derived from basic human–psychology tenets as well as from user–behavior observations.

Chapter V presents an example of how the patterns can be used towards translating system requirements into a user–interface design specification. The relationships between the patterns also were transferred into a network depiction (which is presented in Appendix B).

CHAPTER V

A PROTOTYPE: SAMPLE DESIGN FOR A PORTION OF A DRAWING PROGRAM USER INTERFACE

The set of 30 user–interface patterns (UIPs) delineated in Chapter IV obviously cannot represent a complete pattern language for user–interface design; but the 30 UIPs do serve to illustrate the approach of the pattern–language methodology and the configuration of typical patterns. Earlier it was noted that the patterns form a network, and Appendix B further depicts the dependencies of these patterns.

The present chapter exemplifies that the user–interface pattern language can be practically utilized. From system requirements and functional semantics, a set of patterns are found to represent a user–interface design specification. This prototype illustrates the manner through which some high–level aspects of a drawing–program interface may be quickly designed using a minimal number of patterns.

Step 1: Finding a User–Interface Pattern (UIP) That Best Describes the Project

Before the user–interface pattern language is able to be used, the requirements and functional semantics of the project must be identified. In this example, the basic requirement is that a set of related functions need to be available. For example, the drawing program should allow the user to draw freehand, execute circles and rectangles, etc. In other words, there must be a relatively large number of related functions that will be alternately used. Since a mouse–oriented system is considered as a given, the user will be assumed to typically select functions by using the mouse.

During Step 1, the list/catalog of UIPs must be examined in order to discover the UIP that best describes the overall scope of the project. In other words, the UIP that will be most influential with respect to the user interface must be identified. In some cases, more than one pattern will relevant. For this example, it is assumed that preliminary empirical studies have indicated that the functions mentioned within the preceding paragraph will be the most frequently used (relative to the other functions and options available). In general, utilizing a drawing program involves a large number of tools and tool changes; therefore, methods which minimize the time it takes to select and change tools would be advantageous. The UIP that best fits these requirements is UIP 10: PALETTES. UIP 10: PALETTES is probably the one that most adequately depicts the scope of this small project; and, thus, it will be an integral part of the screen design. In addition, UIP 12: MANIPULABLE WINDOWS also should be included since all the drawing will occur inside windows; as a supplementary facet, the program should allow any number of windows to be open.

Step 2: Starting to Search Through the User–Interface Pattern (UIP) Network

All the UIPs that establish the context for the pattern(s) selected in the Step 1 must be considered; and, if the power to include them exists, then they should be included. In this case, UIP 3: STANDARD SET OF ELEMENTS is not relevant since the current task does not involve defining a set of user–interface elements within a programming community. It is to be assumed that palettes are included in the set of standard elements. In addition, UIP 7: PERSONALIZABLE SCREEN LAYOUT will be included in the project. Within this latter UIP, the user should be allowed to tailor the screen layout, and it is important for the program–state information to not change unless the user requests it.

It is now appropriate to consider the patterns which embellish the pattern(s) selected in the Step 1. First of all, the palette may be used a distinguishing feature by incorporating UIP 4: DISTINGUISHING FEATURES; thus, the palette should be protected from being obstructed by other screen elements through including UIP 13: PROTECTED ELEMENTS. In addition, the functions not included in the palette should be placed into menus which may be accessed via UIP 11: PULL–DOWN MENUS. A palette selection should not change unless the user explicitly changes it; hence, UIP 23: USER IN CONTROL should be incorporated. Finally, the elements on the screen must appear physically separated in order to avoid confusion with respect to the structure of the interface. In this case, the boundaries between the palette, the menu bar, and the windows are relevant.

Step 3: Continuing and Completing the Search Through the User–Interface Pattern (UIP) Network

Step 2 delineated all the UIPs immediately preceding and immediately succeeding each UIP (or UIPs) identified in Step 1. Now, Step 2 must be repeated for all the UIPs that were identified in Step 2. Larger patterns should be included only if the power to enforce them is given. Smaller UIPs should be included only if they do not go beyond the level of detail dictated by the current project. After having cycled through Step 2 for every pattern that is relevant, the following pattern language for this small example emerges:

1. UIP 4: DISTINGUISHING FEATURES
2. UIP 7: PERSONALIZABLE SCREEN LAYOUT
3. UIP 10: PALETTES
4. UIP 11: PULL–DOWN MENUS
5. UIP 12: MANIPULABLE WINDOWS
6. UIP 13: PROTECTED ELEMENTS
7. UIP 18: LEVELS OF VIEW
8. UIP 25: ELEMENT BOUNDARIES
9. UIP 26: WORK AREA TOWARDS THE CENTER
10. UIP 27: FUNCTION SELECTION TOWARDS THE EDGES
11. UIP 28: AUTOSCROLL
12. UIP 30: ABSTRACTIONS WITHOUT RESTRICTIONS

The representational network for the UIPs in the preceding list is portrayed in Figure 27.

The 12 patterns in the preceding list address the most important aspects regarding the presentation of the drawing program to the user. Thus, it could be said that they represent the *facade* of the program. The rough sketch in Figure 28 portrays how the interface may be visualized by drawing out some of the UIPs. Some program functionality is addressed by UIP 18: LEVELS OF

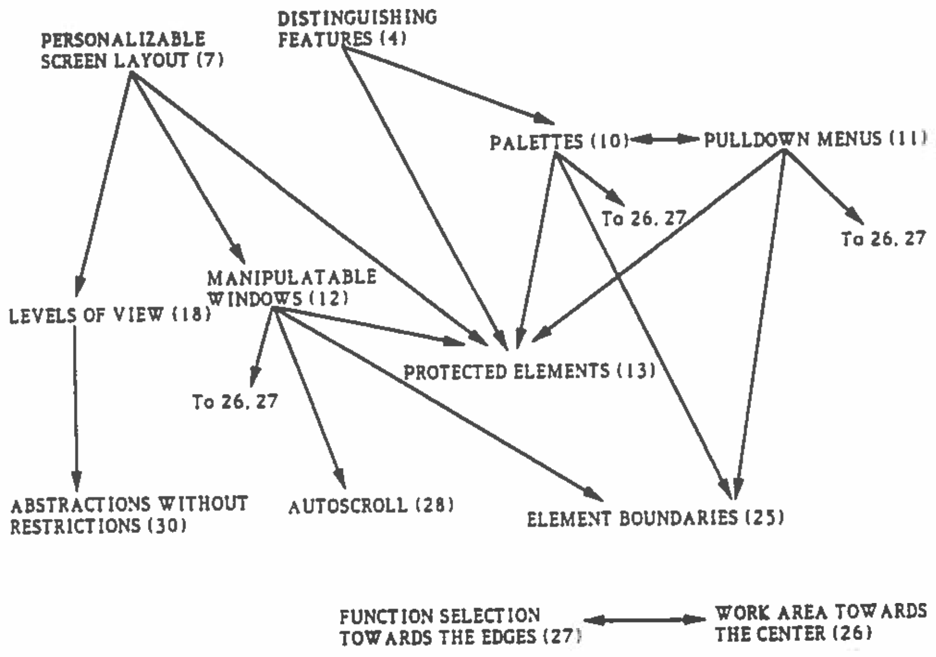

DISTINGUISHING FEATURES (4)
PERSONALIZABLE SCREEN LAYOUT (7)
PALETTES (10)
PULLDOWN MENUS (11)
To 26, 27
To 26, 27
MANIPULATABLE WINDOWS (12)
LEVELS OF VIEW (18)
PROTECTED ELEMENTS (13)
To 26, 27
ABSTRACTIONS WITHOUT RESTRICTIONS (30)
AUTOSCROLL (28)
ELEMENT BOUNDARIES (25)
FUNCTION SELECTION TOWARDS THE EDGES (27)
WORK AREA TOWARDS THE CENTER (26)

FIGURE 27. The Network for the Drawing Interface Patterns

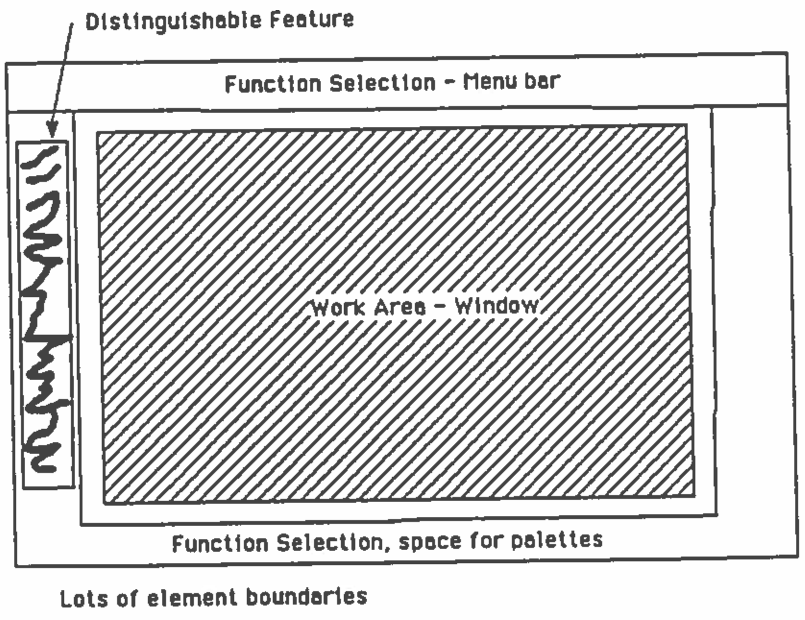

Distinguishable Feature
Function Selection - Menu bar
Work Area - Window
Function Selection, space for palettes
Lots of element boundaries

FIGURE 28. A Rough Sketch of the Facade of a User Interface

VIEW and UIP 30: ABSTRACTION WITHOUT RESTRICTIONS, but these patterns are also closely related to the facade of the program. As is pointed out in Chapter VI, one of the problems with the pattern–language methodology within the user–interface domain is that only a subset of the patterns can be drawn out on paper. For example, it is difficult to draw UIP 30: ABSTRACTION WITHOUT RESTRICTIONS.

Step 4: Adding Personal Patterns

The existing set of UIPs is incomplete. Yet, no matter how many UIPs are eventually defined, there always will be cases in which new and different UIPs are required. In the prototype example, it is already known that the palette will be placed somewhere close to the edges of the work space. However, no UIP exists to indicates whether or not a single palette should be used for all the windows (as opposed to each window being associated with a palette of its own). Therefore, a new UIP is needed. However, this is not a typical *personal* UIP, but rather a UIP that illustrates the incomplete state of the pattern set; given this realization, such a UIP probably should be included within the next general set of UIPs yet to be delineated. In the absence of specifications, it will be assumed that the needed (new) UIP indicates that only one palette should be used (e.g., in SuperPaint) instead of associating a specific palette with each window. (In this instance, UIP 4: DISTINGUISHING FEATURES is preserved because, even though no windows are open, the palette will remain visible.)

Summary

Given a few system requirements and a description of functional semantics, an example may easily be provided to represent how some aspects of a drawing–program user interface could be designed using the patterns previously presented in Chapter IV. Four simple steps, to indicate the proper user–interface patterns which should be identified, completed the necessary priorities instrumental to presenting an integrated example.

The foregoing completes an initial attempt at defining the interrelatedness of a few relevant user–interface patterns. In Chapter VI, preliminary results and derivational limitations of the research that has been conducted are discussed. Comments are presented regarding whether or not the pattern–language approach is feasible. An outline is also provided in order to clarify the various issues which remain to be resolved if the approach ever will be of practical use.

CHAPTER VI

CONCLUSIONS

This thesis is descriptive of efforts towards analyzing the feasibility of a new user–interface design methodology. This new methodology was based on Alexander's (1979) and Alexander et al.'s (1977) work in the architectural domain, as well as on several apparent similarities between architectural design and user–interface design (Hooper, 1986). It has been asserted that the process of translating system requirements and functional semantics into a user–interface design specification can be complemented by employing the pattern–language methodology. A pattern typically addresses (a) a problem that arises due to human factors, (b) the context in which the problem arises, and (c) one or more possible solutions to the problem. The main task of this research was to relate problems observed within the human/computer interaction to the underlying reasons for the problems, and then to find one or more solutions that would reduce (and ideally eliminate) the problems. Most of the patterns thusly unearthed were based on human needs, feelings, perceptions, limitations, and instincts. Ultimately, the research led to the specification of 30 patterns; subsequent to the identification of these 30 UIPs, it was possible to demonstrate how the methodology may be used to produce a user–interface design specification through presenting an example. Although it was beyond the scope of this thesis to develop a comprehensive collection of patterns (and regardless that many claims were made which have yet to be empirically proven), the conclusion arrived at is that the pattern–language approach warrants further investigation.

Results

Through the presentation of an initial set of user–interface patterns (UIPs), a pattern language has been presented. It also have been demonstrated that some aspects of user–interface design can be captured by this specific set of UIPs. The main contribution of this thesis is that it indicates that it is possible to generate user–interface design specifications by using a methodology which incorporates as input the system requirements and functional semantics, and then produces a design specification. This is a considerable step forward from traditional user–interface design guidelines (ACI, 1987; Smith & Mosier, 1986) in which typically all the rules, principles, and postulates are stated as stand–alone facts with no indication as to when, where, and/or why they are relevant. This thesis also represents distinctive step forward, relative to previously existing design methodologies that address the translation process. For example, Moran's (1981b) *command–language grammar* produced a virtually infinite number of design specifications, whereas the methodology proposed herein is able to produce a more detailed and categorical specification.

The Usefulness of User–Interface Patterns (UIPs)

One of the most useful aspects of the pattern–language approach is that it ties together a large number of individual and fragmented pieces of knowledge.

Experts in user–interface design have typically adhered to a coherent set of principles and experiences that they are able to incorporate into a single specification; these experts know how the various rules relate to and effect one another, and they are able to identify problem situations and create design alternatives. Less experienced designers, on the other hand, may actually be aware of the same principles as are the experts; yet the less–experienced may lack the a complete understanding regarding how the principles are interrelated. Nevertheless, the problem is not to learn the individual rules; it is, rather, to understand when the rules apply, how they can be applied, and how they relate to each other. And, moreover, it must be recognized that each UIP is an attempt to describe these parameters.

UIPs specify when they apply by describing particular situations; in other words, UIPs are context sensitive. UIPs describe how they can be satisfied either (a) by giving specific solutions regarding how the interface should be designed or (b) by referring to smaller and more specific UIPs. Thus, a UIP describes its relationships with other UIPs by specifying which other (larger) UIPs are needed for it to be present, and also by specifying the (smaller) UIPs that are needed for it to be complete. The fact that only 30 UIPs have been defined denotes that the existing user–interface pattern language is neither morphologically nor functionally complete.

Towards Morphological Completeness

As it may be recalled, a pattern language is morphologically complete if and only if its UIPs form a complete interface within which all the details have been addressed. The UIPs defined in this thesis address only a representative (but small) subset of all the elements which constitute a user interface and only a small portion of the design issues that may be involved. Additional UIPs unquestionably are needed. Within the UIP definitions (contained in Chapter IV), potential yet–to–be–developed UIPs were suggested. For example, UIPs that address high–level issues (such as direct manipulation) and UIPs that address low–level issues (such as the ordering and grouping of menu items) are needed.

A *general* pattern language, in itself, never will be morphologically complete. (This fact introduces the inherent need for project–specific UIPs which are discussed in the section entitled Special–Purpose Projects).

Towards Functional Completeness

A language is functionally complete if and only if the patterns in the language resolve all the conflicting forces which may possibly arise. The set of UIPs delineated within this thesis does not resolve all possible conflicting forces. Once again, the fact remains: There simply is not enough UIPs. However, the ultimate question still arises: If there were an extensive number of general UIPs, would it be possible to know absolutely that the language was even close to functionally complete? The only way to answer this question would be to evaluate the user–interface designs with respect to a set of performance measures (refer to Table 1, in Chapter I, on p. 8). In the author's opinion, functional completeness is inconceivable. Functional completeness specifically implies that the human/computer interaction would be flawless; but, as emphasized by UIP 21: WARNING THE USER, *slips* are impossible for the machine to detect. Thus, accomplishment of perfect functional completeness appears to be, at least at this time, to be impossible. The conflicting forces which may be present include (a) the existence of actions which are irreversible, (b) the system which may assume that the user will respond *correctly* to warnings about irreversible actions, and (c) the users who sometimes make unexplainable mistakes. These forces serve as a counterexamples which invalidate the claim that all conflicting forces in the human/computer interaction can be resolved. Unless it is possible to assume that the user is perfect with respect to his or her actions, there will always exist conflicting forces that cannot be eliminated.

Special–Purpose Patterns

No matter how many UIPs that are eventually defined, there will always be special cases and unique projects for which no existing UIPs apply. Consequently, it is important to distinguish between general (project–independent) UIPs (such as those defined in this thesis and those which Alexander et al., 1977, defined) versus UIPs that need to be specifically defined for a particular project in order to meet the special requirements set forth by a user.

Alexander et al.'s (1977) general set of patterns addresses the most common issues which arise and the decisions which must be made in architectural design. However, even within the field of architecture, there may be projects in which the client has specific requirements that are not addressed by the existing patterns. In such cases, the architect attempts to specify exactly what the client requires by creating new patterns; and, such new patterns are specific to that particular client and project. A useful example is Alexander et al.'s (1968) pattern language for a multiservice center (i.e., for a community facility that provides a variety of special services to citizens). The multiservice center described by Alexander et al. (1968) was intended to help solve the problems that arise in low–income communities; and, a specialized set of patterns had to be designed for this particular project.

Scenarios similar to that presented by the multiservice center are likely to present themselves in the domain of user interfaces. A general set of UIPs probably will be sufficient for addressing most issues in a large number of user–interface designs; moreover, the need for some project–dependent UIPs probably will also arise. This is especially true when designing user interfaces for particular user groups (e.g., for users with poor eyesight or users with a handicap that does not allow them to utilize the regular input and output devices). In such individualized instances, the designer must identify the needs of the users and the conflicts that must be resolved; and, then, the designer needs to develop patterns that eliminate the conflicts. There will always be special projects that require customized UIPs.

<u>Discussion</u>

Integrating Users Within the Design Process

The pattern–language methodology assumes that the users of a new system will contribute to the user–interface design process from the very beginning. Two reasons may be provided to substantiate this recommendation. First, the methodology translates system requirements and functional semantics into a user–interface design specification; in other words, the design specification is highly dependent on the nature of the tasks the system should be able to carry out. The design specification also depends on how the functions of the system should work. If the purpose is to design a user interface that satisfies user requirements, it is simply sensible for users to be integrated into the process at its initial stages. Second, a general pattern language will never suffice for a large project; specialized patterns also will need to be created as well. These specific patterns are based upon the users of the system, not on the designers or programmers.

Top–Down Versus Bottom–Up Design

The translation process, from system requirements and functional semantics to a user–interface design specification, produces a series of UIPs (an example of which was provided in Chapter V). After the UIPs have been identified, they must be implemented; that is, the UIPs must be satisfied within the implementation of the user interface. The nature of pattern languages suggests that a top–down approach should be undertaken because the larger UIPs must be present in order for the smaller UIPs to exist. However, a strict top–down approach does not seem to be appropriate in the process of implementing or satisfying patterns in practice.

In the architectural domain, it has been observed that a design unfolds bottom–up simultaneously with the high–level ideas and concept which are kept in mind. According to G. Z. Brown (personal communication, May 1990), in practice, there is a mixture of the top–down and bottom–up approach. Thus, it may be unrealistic to assume that a user–interface pattern language can be used as a basis for a strictly top–down implementation process.

Aesthetics

User–interface patterns (UIPs) are useful for increasing the awareness of the regarding the relationships among different aspects of user–interface design. The UIPs explain where, when, and why certain problems occur and how they can be solved; they provide instructions regarding how user–interface elements should be positioned on the screen; and they suggest the type of functionality which should be provided in order to satisfy user requirements. Yet, the element that is missing in all this is *aesthetics*. This deficiency is well–known in the architectural domain. Patterns can aid the designer to address all the issues and problems that arise in the design process, but the implementation of the patterns does not necessarily result in an aesthetically pleasing structure. Aesthetics is also an important aspect of user–interface design. A pattern language for user interfaces will suffer from the same shortcomings that have been recognized in the architectural domain. According to G. Z. Brown (personal communication, May 1990), this problem is a major obstacle inhibiting the complete automation of the design process.

Refining UIP Interrelationships

For each UIP defined in this thesis, a related set of smaller and larger UIPs has been specified in the same manner as Alexander et al. (1977) itemized related patterns. The difference which exists in the domain of user–interface design concerns the fact that, for every single pattern, the number of related patterns seems to be much higher. In other words, there seems to exist user–interface patterns which should and could be used to embellish all other patterns. For example, UIP 20: REVERSIBLE ACTIONS, UIP 24: EXPLAINING MESSAGES, and UIP 29: SHORTCUTS FOR THE EXPERT can be considered as embellishing patterns for most of the other (both large and small) patterns in the language. In the domain of user–interface design, it is apparent that there exists some *global* set of patterns which apply in all aspects of the design. If such global patterns exist, it must be determined how they can be elegantly included in the language without making them just as vague as traditional user–interface guidelines.

User–Interface Patterns (UIPs) Must Be Evaluated

Finally, user interfaces must be evaluated; and, likewise, user–interface patterns also must be evaluated. This is another reason why the users should constitute an integral component of user–interface development. A set of UIPs will be of no use unless each pattern has been empirically tested. Individual experiments probably will have to be individually conducted for each UIP. Experiments that will test an individual UIP and produce results that generate information with respect to a set of performance measures must be developed.

It was beyond the scope of this thesis to evaluate the UIPs. The focus was directed towards whether it could be possible to form a pattern language given that the patterns were be validated; that is, the goal was to validate the UIPs.

Nevertheless, it should be noted that G. Z. Brown (personal communication, May 1990) concluded that, even though Alexander et al. (1977) attempted to justify their patterns, it is generally recognized (within the architectural field) that it is essentially impossible to measure the goodness of many designs and design decisions.

Drawable Versus Nondrawable User–Interface Patterns (UIPs)

Alexander (1979) argues that, if structural relationships cannot be captured in a drawing, then they do not constitute a pattern. However, it does not seem reasonable for such an example or an illustration to be required for user–interface patterns. The dissimilarity involves the fact that dynamics do not need to be considered in architectural design to the same extent that they must be considered in user–interface design. To be more specific, user–interface design requires one or more patterns which address the need for viewing data at different levels of abstraction; and, furthermore, the level of abstraction should not impose any restrictions with respect to functionality. Such user–interface patterns are more appropriately depicted using storyboards. The question concerns, however, whether or not it is possible to develop a more elegant way of translating user–interface patterns.

Automating the Pattern–Language Methodology

Given a set of patterns and a process for manipulating the patterns, an interesting question comes to mind: Is it possible to automate the translation of requirements and functional semantics into a user–interface design specification? The answer is both yes and no; there are interesting aspects of the design process that can be automated and there are aspects that cannot be automated.

It is seems reasonable to first examine the aspects that cannot be automated. A preceding discussion emphasized that UIPs neglect design aesthetics. In essence, the problem is that automating design creation in a manner which would be pleasing to the eye is beyond current technology. Project–specific patterns also hinder the possibility of completely automating the translation process. Even though the process of creating and identifying patterns might somehow be supported by an expert system, patterns are based on user requirements; and, consequently, a considerable amount of user participation is necessary.

In the author's opinion, the translation process might be automated (at least partially) to the extent that a design specification similar to the one in Figure 28 could be produced (refer to Chapter V, p. 131).

First, each high–level requirement specified by the user would need to be matched with a UIP. The example in Chapter V illustrated that the need for frequent mode changes resulted in the selection of UIP 10: PALETTES. To automate this mapping, a great deal of knowledge must be associated with every specific UIP. For example, in this case, the condition that the palette is effective for frequent and rapid mode changes must be encoded. Knowledge about the shape and size of the archetypical palettes also must be available.

Second, once a requirement has been mapped into a UIP, a production system might determine the related (larger and smaller) UIPs. As the UIPs become smaller, more information about the design specification is generated. For example, if the production system accepted UIP 10: PALETTES as input, it could return a hierarchy similar to the one illustrated in Figure 27 (refer to Chapter V, p. 131). Such an automated design program would then not only have knowledge about archetypical palettes, but also would know that the palette should be located towards the edges of the screen since it is an instance of a function–selection element (and, thus, subject to UIP 27: FUNCTION SELECTION TOWARDS THE EDGES).

Even though the scheme described in the preceding is conceivable, there are several difficult problems that require resolution. Although it is possible that the scheme could work relatively well for small projects with few high–level requirements (such as the one in Chapter V), problems would arise if many (possibly conflicting) requirements needed to be considered (unless specific patterns for these conflicting requirements existed). For example, following satisfaction of Requirement A, removal of Requirement A may be required in order to satisfy Requirement B. In a second attempt to satisfy Requirement A, Requirement B may need to be removed. Consequently, the system would need to know about tradeoffs, and how to tweak UIPs and requirements so that all the requirements may be met. In reality, project–specific patterns probably would have to be individually created. Humans are relatively good at solving such problems, but automating these processes is difficult.

In addition, problems related to knowledge representation would have to be solved. Consider, for example, UIP 10: PALETTES and UIP 27:

FUNCTION SELECTION TOWARDS THE EDGES. It has been previously established that knowledge regarding an archetypical palette would need to be associated with UIP 10: PALETTES. Although there is no logical reason why knowledge about the position of a typical palette should be deferred to UIP 27: FUNCTION SELECTION TOWARDS THE EDGES, given the way the pattern language is organized this is exactly what happens. The question is only reasonable: Why is not knowledge about the shape and size of a palette deferred to lower–level palettes? Presumably, there should exist patterns to address the possible shapes of palettes. However, the point is that the pattern language must be made consistent with respect to the levels at which the information (of varying detail) is introduced.

In conclusion to this section, it must be emphasized that––before an attempt is made to automate the translation process––a better understanding is needed with respect to how the translation process is carried out by the human designers for the human users.

Some Unexplored Areas

In conformance with its intention, this thesis embraced a very limited scope of patterns. For example, few patterns directly address the high–level requirements which concern direct manipulation or none zero–in on metaphors. For example, there currently do not exist UIPs which address such notions as dragging or how to indicate that an object is touching or connected to another object. Neither are low–level patterns included to address how icons should look or how menu items should be ordered. In part this is due to the facts that the pattern–language approach does not minimize the problems encountered in the resolution of such complex issues, and generating UIPs to address these matters is very difficult. It may be remembered that UIPs should ultimately bottom out with specific suggestions regarding how a problem may be solved. The type and extent of the metaphors are project dependent, and probably cannot be captured by general UIPs. This seems to be one of the many areas in which project–dependent UIPs must be created.

In order to estimate the number of UIPs which would constitute a useful set for a user–interface designer, the power of the language (i.e., the number of patterns) versus ease of the language must be considered. As the power increases, the ease of use decreases. Clearly, there exists more than the 30 UIPs which were defined herein. Throughout the process of defining even these 30 UIPs (in Chapter IV), approximately 15 potentially additional UIPs and areas of development were specified. Through extrapolation, it is estimated that there exists a minimum of 50 patterns; it is more difficult to determine an upper bound for the number of patterns. Nevertheless, any general set of UIPs should describe similarities which are apparent within the world of user–interface design. A general UIP should not describe elements which only can be attested to within a single design. This greatly limits the potential number of UIPs that may be present in a general pattern language for user–interface design. In the author's opinion, the number of UIPs in a general pattern language probably will not, and should not, exceed approximately 500. If the number of patterns greatly exceeded 500, the language might become difficult and impractical in terms of utilization. The number of UIPs also must be limited in order for the language to be manageable. Experiences with the Macintosh toolbox (which contains 900 toolbox and operating–system routines) and large programming languages (such as ADA, with approximately 200 reserved words, predefined attributes, pragmas, operators, and types) indicate that a tool can become difficult to learn and to use if the number of entities that the user must be concerned with becomes too large.

Final Note

This thesis serves as an inspiration to explore new ways of translating system requirements and functional semantics into a user–interface design specification. Traditional user–interface guidelines provide negligible assistance in this process, and presently there exists no acceptable method by which graphical interfaces of event–driven systems may be represented. The presentation, within this thesis, of a user–interface design methodology is an approach which necessitates further development before any final conclusions regarding its viability can be posited; yet, the approach does seem to be promising. The anticipation is that others will be inspired by this work and that these others will begin to view user–interface designs in terms of patterns. Identifying, developing, and evaluating user–interface patterns requires a significant amount of work; but a collective effort among researchers could lead to a user–interface design methodology, as well as to a set of patterns which would approach both morphological and functional completeness.

APPENDIX A

LEARNING ABOUT PATTERNS AND PATTERN LANGUAGES

LEARNING ABOUT PATTERNS AND PATTERN LANGUAGES

In the initial phases of this thesis study, it became evident that the notion of patterns and pattern languages required an expanding understanding. Specifically, there were two important questions necessitated answers. How are pattern languages typically used in real–life projects? How are project–specific patterns typically created? Since the answers to these questions could not be found in textbooks, interviews were conducted with an architect who utilizes the pattern–language approach in order to achieve a fuller understanding of the relevant solutions. During one of these interviews, the architect and the interviewer (thesis author) were recorded by one video camera, while a second camera captured the illustrations drawn by the architect. The answers to the initial two questions posed at the beginning of this paragraph were addressed.

Using a Pattern Language in a Real–Life Project

The major revelation from the interviews was a comprehension of how pattern languages are used in practice. The most important point was that the clients (of the particular field/discipline involved) should be involved in the design process from the very beginning. According to the available information, one or more architects would typically meet with the client(s); this enables the architect(s) to develop a sense of the physical and social requirements being requested. Although some requirements may be mapped onto already–existing general patterns, usually the existing patterns are used merely as building blocks for the project–specific patterns developed based on particular requirements.

Clients often have difficulties in expressing their desires; in fact, at time, even they do not know what they want. One specific project which was mentioned concerned the University of Massachusetts (in Amherst). The School of Education had developed a new theory of elementary education, and a group of architects was presented with the task to design a new school that would reflect the new theory. The architects needed the developers of the theory to describe the school as they conceived it. But the developers were unable to do so; their inability was due primarily to the fact that images of conventional school designs (which they wanted to abandon) hindered them from imagining new directions.

The architects were required to be more clever in the way they approached the clients. They asked more specific questions such as "Can you describe a typical day in the life of a student?" and "Can you describe a typical day in the life of a teacher?" Over 1–week period, the architects obtained an overall sense of the structure of the school. In addition, the week was very helpful for the clients in that they gained a better understanding of the physical and social requirements of the new school.

After a series of meetings between the architect(s) and the client(s), an architect is able to develop a pattern language that is quite specific to the project and the requirements of the client(s). The language should provide an appropriate sense regarding the design. But, because it does not physically specify the design, the plans for a building may not have been drawn yet; yet, the patterns that will constitute the building(s) provide both the architects and the clients with an adequate idea regarding how the finished design will appear.

When the clients are satisfied with the patterns that have been developed and the architects believe that they have captured all the aspects of the project, the process of integrating the patterns into a complete design is initiated. The result of this process is a specification from which a structure may be built. Thus, the process is comparable: Patterns are used as tools to translate the user requirements into a specification, from which a structure may be designed and then built.

The Creation of a Project Specific Pattern

When architects converse with a client they attempt to attain a sense of the client's physical and social requirements. It cannot be assumed that these requirements are nonconflicting. The architect must attempt to create one or more patterns that will incorporate all the requirements. The following example illustrates that a pattern can be created to accommodate opposing needs.

The clients in this example were founding a new settlement in the north of Israel. As usual, the architects worked with the clients in order to obtain an understanding of the latter's goals. One of the most important issues for the clients was the relationship within the community. On the one hand, they wanted to be very close as a community. They defined this to be the ability to drop in and visit each other often; moreover, within the community, there were smaller groups who were even more solidified. On the other hand, the people in the community wanted to be able to walk out of their houses and immediately appreciate the surroundings *without* coming into contact with other people in the community. These two opposing needs were solved by the pattern depicted in Figure 29. (The figure is a re–creation of the pattern that the architect actually drew on paper during the interview.)

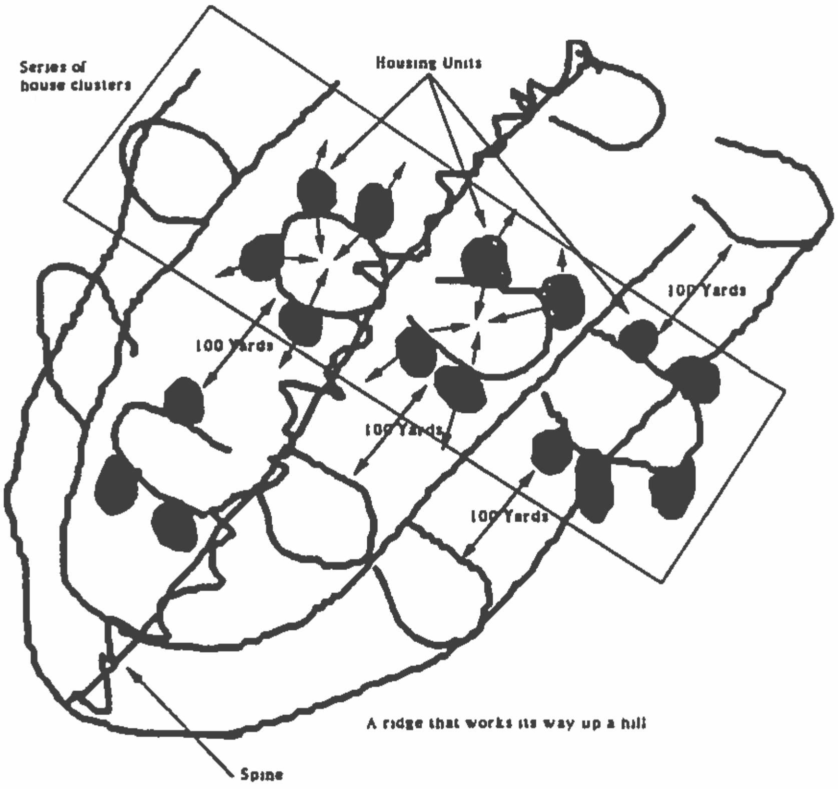


FIGURE 29. A Pattern Satisfying Two Conflicting Requirements

Since the site was on a ridge that worked its way up a hill, the pattern which emerged incorporated a series of house clusters that would come off of small nodes. Thus, a series of houses would be clustered around a bit of common land, then another cluster 100 yards up or down the hill would appear. The closeness of the community was preserved by placing the house clusters close to each other (i.e., within 100 yards). The smaller groups could settle in particular clusters; and, even in particular nodes, within a cluster. Between the clusters, open land existed which helped in terms of satisfying the second requirement.

Even though this example beautifully illustrates how a pattern may be created to satisfies a set of (conflicting) requirements, the interviews did not clarify the steps that the architect(s) went through in order to create the pattern(s). In order to capture this process, a protocol analysis of architects during pattern creation must be conducted. Even with such an analysis, it probably would be difficult to understand exactly how patterns such as the preceding one are created. Creativity is a human phenomenon that is not well–understood.

APPENDIX B

THE USER–INTERFACE PATTERNS (UIPs) ORGANIZED IN A NETWORK

THE USER–INTERFACE PATTERNS (UIPs) ORGANIZED IN A NETWORK

Figure 30 depicts the relationships between the 30 patterns that were defined in Chapter IV. The arrow directed from UIP A to UIP B indicates that UIP A is embellished by UIP B. All the relationships that are depicted within the figure are also specified as a part of the pattern definitions included in Chapter IV. Although the figure is of little practical use to a designer, it serves to illustrate a conceptualization of how the UIPs are interrelated.

A summary of the previously delineated user–interface patterns (UIPs) is provided, not only to facilitate recognition in terms of Figure 30, but also to refresh the reader's memory regarding the UIPs which have been addressed and included within Chapter IV:

1. UIP 1: INDEPENDENT PROGRAMS
2. UIP 2: ACTIVITY CENTER
3. UIP 3: STANDARD SET OF ELEMENTS
4. UIP 4: DISTINGUISHING FEATURES
5. UIP 5: TRANSFER OF DATA
6. UIP 6: EASY TRANSITION BETWEEN PROGRAMS
7. UIP 7: PERSONALIZABLE SCREEN LAYOUT
8. UIP 8: LIMITED NUMBER OF COMMANDS
9. UIP 9: STANDARDIZED METHODS

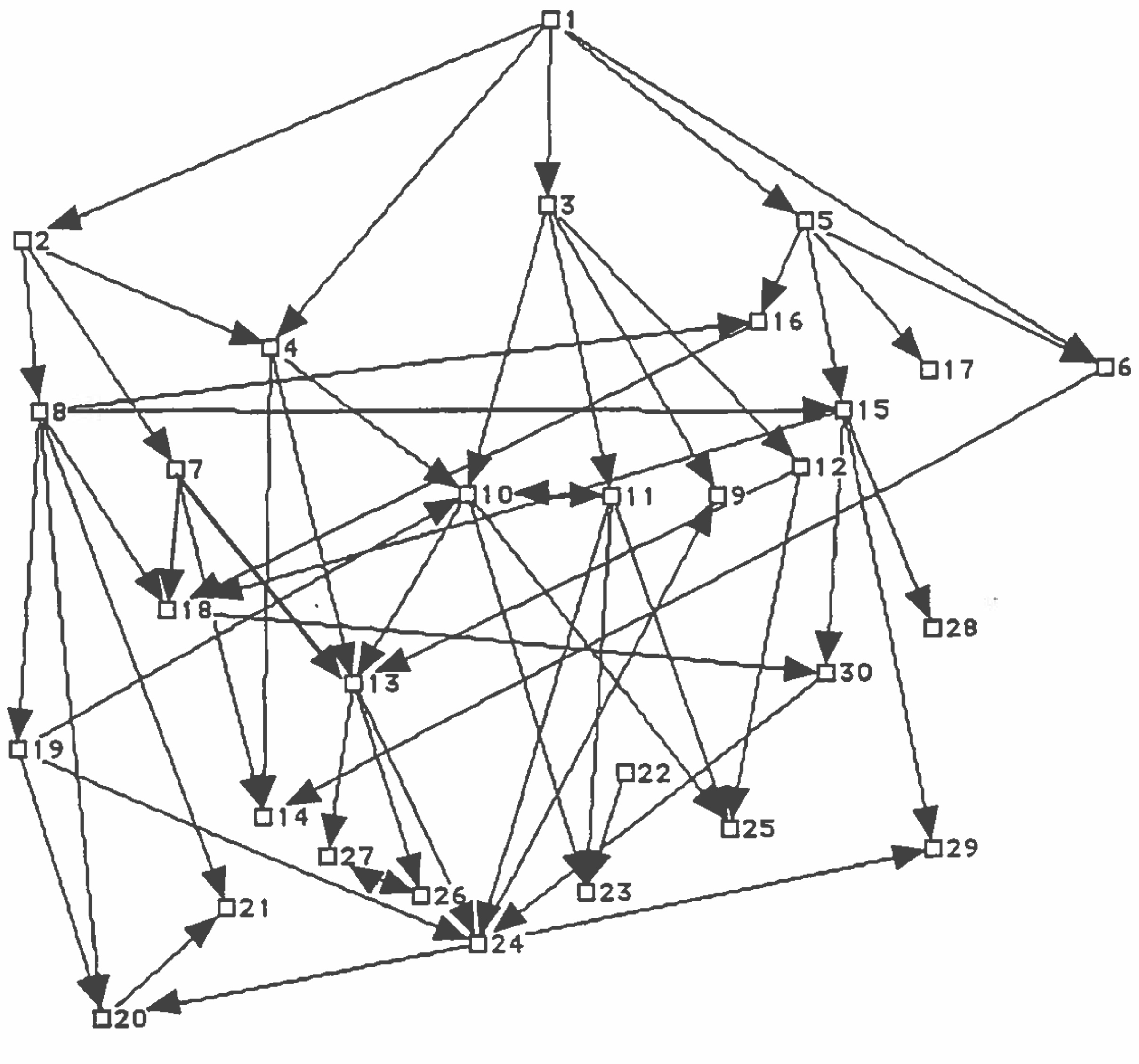


FIGURE 30. The Relationships Between User–Interface Patterns (UIPs)

REFERENCES

Alexander, C. (1979). The timeless way of building (Vol. 1). New York: Oxford University Press.

Alexander, C., Ishikawa, S., & Silverstein, M. (1968). A pattern language which generates multi-service centers. Berkeley, CA: Center for Environmental Structure.

Alexander, C., Ishikawa, S., & Silverstein, M. (1977). A pattern language: Towns, buildings, construction (Vol. 2). New York: Oxford University Press.

Alexander, C., Silverstein, M., Angel, S., Ishikawa, S., & Abrams, D. (1975). The Oregon experiment (Vol. 3). New York: Oxford University Press.

Apple Computer Inc. (1987). Human interface guidelines: The Apple desktop interface. Reading, MA: Addison-Wesley.

Chapman, D. (1981, November). A program testing assistant (AI Memo No. 651). Cambridge: Massachusetts Institute of Technology, Artificial Intelligence Laboratory.

Deatherage, B. H. (1972). Auditory and other sensory forms of information presentation. In H. P. Van Cott & R. G. Kinkade (Eds.), Human engineering guide to equipment design (rev. ed., pp. 123-160). Washington, DC: U.S. Government Printing Office.

Furnas, G. W., Landauer, T. K., Gomez, L. M., & Dumais, S. T. (1987). The vocabulary problem in human-system communication [Research contributions]. Communications of the ACM, 30, 964-971.

Grether, W. F., & Baker, C. A. (1972). Visual representation of information. In H. P. Van Cott & R. G. Kinkade (Eds.), Human engineering guide to equipment design (rev. ed., pp. 41-121). Washington, DC: U.S. Government Printing Office.

Hooper, K. (1986). Architectural design: An analogy. In D. A. Norman & S. W. Draper (Eds.), User centered system design (pp. 9-23). Hillsdale, NJ: Erlbaum.

Hutchins, E. L., Hollan, J. D., & Norman, D. A. (1986). Direct manipulation interfaces. In D. A. Norman & S. W. Draper (Eds.), User centered system design (pp. 87-124). Hillsdale, NJ: Erlbaum.

Jacob, R. J. K. (1983). Using formal specifications in design of a human-computer interface [Research contributions]. Communications of the ACM, 26, 259-264.

Johnson-Laird, P. N. (1983). Mental models. Cambridge, England: Cambridge University Press.

Moran, T. P. (1981a). An applied psychology of the user [Guest editor's introduction]. Computing Surveys, 13(1), 1–11.

Moran, T. P. (1981b). The command language grammar: a representation for the user interface of interactive computer systems. International Journal of Man–Machine Studies, 15, 3–50.

Smith, S. L., & Mosier, J. N. (1986, August). Guidelines for designing user interface software (Tech. Rep. No. ESD–TR–86–278). Bedford, MA: The Mitre Corporation. (Available from National Technical Information Service, 5285 Port Royal Road, Springfield, VA 22161, NTIS Doc. No. AD A177 198)