

Achieving Usability Through Software Architecture

Len Bass
Bonnie E. John
Jesse Kates

June 2001

TECHNICAL REPORT
CMU/SEI-2001-TR-005
ESC-TR-2001-005



Carnegie Mellon
Software Engineering Institute

Pittsburgh, PA 15213-3890

Achieving Usability Through Software Architecture

CMU/SEI-2001-TR-005

ESC-TR-2001-005

Len Bass
Bonnie E. John
Jesse Kates

June 2001

Architecture Tradeoff Analysis Initiative

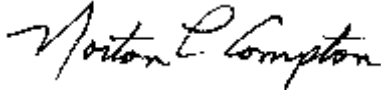
Unlimited distribution subject to the copyright.

This report was prepared for the

SEI Joint Program Office
HQ ESC/DIB
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER



Norton L. Compton, Lt Col., USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

Copyright 2001 by Carnegie Mellon University.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

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

Internal use. Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and "No Warranty" statements are included with all reproductions and derivative works.

External use. Requests for permission to reproduce this document or prepare derivative works of this document for external and commercial use should be addressed to the SEI Licensing Agent.

This work was created in the performance of Federal Government Contract Number F19628-00-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (<http://www.sei.cmu.edu/publications/pubweb.html>).

IBM, ViaVoice, Adobe, PhotoShop, Microsoft are registered trademarks of their respective organizations.

Table of Contents

Abstract	xi
1 Introduction	1
2 General Usability Scenarios	5
2.1 Aggregating Data	5
2.2 Aggregating Commands	5
2.3 Canceling Commands	5
2.4 Using Applications Concurrently	6
2.5 Checking for Correctness	6
2.6 Maintaining Device Independence	6
2.7 Evaluating the System	6
2.8 Recovering from Failure	6
2.9 Retrieving Forgotten Passwords	7
2.10 Providing Good Help	7
2.11 Reusing Information	7
2.12 Supporting International Use	7
2.13 Leveraging Human Knowledge	8
2.14 Modifying Interfaces	8
2.15 Supporting Multiple Activities	8
2.16 Navigating Within a Single View	8
2.17 Observing System State	9
2.18 Working at the User's Pace	9
2.19 Predicting Task Duration	9
2.20 Supporting Comprehensive Searching	9
2.21 Supporting Undo	10
2.22 Working in an Unfamiliar Context	10
2.23 Verifying Resources	10
2.24 Operating Consistently Across Views	10
2.25 Making Views Accessible	11
2.26 Supporting Visualization	11
3 Details of the Usability Benefit Hierarchy	13
3.1 Increases Individual User Effectiveness	14
3.1.1 Expedites routine performance	14

	Accelerates error-free portion of routine performance	15
	Reduces the impact of routine user errors (slips)	15
3.1.2	Improves non-routine performance	15
	Supports problem-solving	15
	Facilitates learning	16
3.1.3	Reduces the impact of user errors caused by lack of knowledge (mistakes)	16
	Prevents mistakes	17
	Accommodates mistakes	17
3.2	Reduces the Impact of System Errors	17
3.2.1	Prevents system errors	18
3.2.2	Tolerates system errors	18
3.3	Increases user confidence and comfort	18
4	Categorization By Usability Benefit	19
4.1	Aggregating Data	20
4.2	Aggregating Commands	20
4.3	Canceling Commands	22
4.4	Using Applications Concurrently	22
4.5	Checking for Correctness	23
4.6	Maintaining Device Independence	25
4.7	Evaluating the System	25
4.8	Recovering From Failure	26
4.9	Retrieving Forgotten Passwords	26
4.10	Providing Good Help	27
4.11	Reusing Information	27
4.12	Supporting International Use	28
4.13	Leveraging Human Knowledge	29
4.14	Modifying Interfaces	31
4.15	Supporting Multiple Activities	31
4.16	Navigating Within a Single View	32
4.17	Observing System State	32
4.18	Working at the User's Pace	33
4.19	Predicting Task Duration	34
4.20	Supporting Comprehensive Searching	35
4.21	Supporting Undo	35
4.22	Working in an Unfamiliar Context	36
4.23	Verifying Resources	37
4.24	Operating Consistently Across Views	37
4.25	Making Views Accessible	38

4.26	Supporting Visualization	39
5	Software Engineering Hierarchy	41
5.1	Separation	42
5.1.1	Encapsulation of function	42
5.1.2	Data from commands	42
5.1.3	Data from the view of that data	42
5.1.4	Authoring from execution	43
5.2	Replication	43
5.2.1	Data	43
5.2.2	Commands	44
5.3	Indirection	44
5.3.1	Data	44
5.3.2	Function	44
5.4	Recording	45
5.5	Preemptive Scheduling	45
5.6	Models	46
5.6.1	Task	46
5.6.2	User	46
5.6.3	System	46
6	Architectural Patterns and Categorization	47
6.1	Aggregating Data	47
6.1.1	Pattern	47
6.1.2	Allocation to mechanism hierarchy	49
6.2	Aggregating Commands	49
6.2.1	Pattern	49
6.2.2	Allocation to mechanism hierarchy	51
6.3	Canceling Command	51
6.3.1	Pattern	51
6.3.2	Allocation to mechanism hierarchy	54
6.4	Using Applications Concurrently	55
6.4.1	Pattern	55
6.4.2	Allocation to mechanism hierarchy	55
6.5	Checking for Correctness	56
6.5.1	Pattern	56
6.5.2	Allocation to mechanism hierarchy	57
6.6	Maintaining Device Independence	57
6.6.1	Pattern	57
6.6.2	Allocation to mechanism hierarchy	58
6.7	Evaluating the System	58
6.7.1	Pattern	58
6.7.2	Allocation to mechanism hierarchy	59

6.8	Recovering from Failure	59
6.8.1	Pattern	60
6.8.2	Allocation to mechanism hierarchy	60
6.9	Retrieving Forgotten Passwords	61
6.9.1	Pattern	61
6.9.2	Allocation to mechanism hierarchy	61
6.10	Providing Good Help	61
6.10.1	Pattern	61
6.10.2	Allocation to mechanism hierarchy	62
6.11	Reusing Information	63
6.11.1	Pattern	63
6.11.2	Allocation to mechanism hierarchy	64
6.12	Supporting International Use	64
6.12.1	Pattern	64
6.12.2	Allocation to mechanism hierarchy	66
6.13	Leveraging Human Knowledge	66
6.13.1	Pattern	66
	Platform standards	66
	Multiple interfaces	66
6.13.2	Allocation to mechanism hierarchy	67
6.14	Modifying Interfaces	67
6.14.1	Pattern	67
6.14.2	Allocation to mechanism hierarchy	67
6.15	Supporting Multiple Activities	68
6.15.1	Pattern	68
6.15.2	Allocation to mechanism hierarchy	68
6.16	Navigating Within a Single View	69
6.16.1	Pattern	69
6.16.2	Allocation to mechanism hierarchy	69
6.17	Observing System State	69
6.17.1	Pattern	70
6.17.2	Allocation to mechanism hierarchy	70
6.18	Working at the User's Pace	71
6.18.1	Pattern	71
6.18.2	Allocation to mechanism hierarchy	71
6.19	Predicting Task Duration	72
6.19.1	Pattern	72
6.19.2	Allocation to mechanism hierarchy	72
6.20	Supporting Comprehensive Searching	72
6.20.1	Pattern	73
6.20.2	Allocation to mechanism hierarchy	73
6.21	Supporting Undo	74
6.21.1	Pattern	74

6.21.2	Allocation to mechanism hierarchy	75
6.22	Working in an Unfamiliar Context	75
6.22.1	Pattern	75
6.22.2	Allocation to mechanism hierarchy	75
6.23	Verifying Resources	76
6.23.1	Pattern	76
6.23.2	Allocation to mechanism hierarchy	76
6.24	Operating Consistently Across Views	76
6.24.1	Pattern	77
6.24.2	Allocation to mechanism hierarchy	78
6.25	Making Views Accessible	78
6.25.1	Pattern	78
6.25.2	Allocation to mechanism hierarchy	78
6.26	Supporting Visualization	78
6.26.1	Pattern	79
6.26.2	Allocation to mechanism hierarchy	79
7	Cross-Referencing Benefits and Mechanisms	81
8	Further Work	83
	References/Bibliography	85

List of Figures

Figure 1. Usability is One Attribute of System Design Among Many	1
Figure 2. Aggregation of Data Architecture Pattern	47
Figure 3. Authoring of Aggregation of Commands Architecture Pattern	50
Figure 4. Module View of Cancellation Architecture Pattern	52
Figure 5. Cancellation Pattern - Thread View	53
Figure 6. Module View of Concurrent Application Use	55
Figure 7. Correctness	56
Figure 8. Virtual Device Layer	58
Figure 9. Data Recording	59
Figure 10. Perform Checkpoint	60
Figure 11. Context Dependent Help	62
Figure 12. Information Re-Use	64
Figure 13. Internationalization	65
Figure 14. Navigation	69
Figure 15. Search	73
Figure 16. Undo	75
Figure 17. Consistent Operation	77
Figure 18. Benefit and Mechanism Matrix	82

List of Tables

Table 1. Usability Benefits Heirarchy	14
Table 2. Usability Benefits Heirarchy	19
Table 3. Software Engineering Hierarchy	41

Abstract

In this report, we present an approach to improving the usability of software systems by means of software architectural decisions. We identify specific connections between aspects of usability, such as the ability to "undo," and software architecture. We also formulate each aspect of usability as a scenario with a characteristic stimulus and response. For every scenario, we provide an architecture pattern that implements its aspect of usability. We then organize the usability scenarios by category. One category presents the benefits of these aspects of usability to users or their organizations. A second category presents the architecture mechanisms that directly relate to the aspects of usability. Finally, we present a matrix that correlates these two categories with the general scenarios that apply to them.

The information in this report can benefit software architecture designers and evaluators. Evaluators can use the scenarios as a checklist to determine whether a particular architecture supports necessary usability features. Designers can use the information in three fashions:

1. The scenarios can serve as a checklist to show whether important usability features have been considered in the requirements.
2. The architecture patterns can help guide the designer in supporting the scenarios.
3. The matrix enables a designer to determine what additional aspects of usability can be supported for minimal cost, since the necessary mechanisms, at least potentially, are in place.

1 Introduction

The goal of this work is to achieve better system usability through design decisions embodied in the software architecture. These decisions are the most difficult to change as the design progresses through the life cycle. Hence, understanding the relationship between software architecture and usability is important to ensure that the system ultimately achieves it. Equally important, however, is expressing this relationship in terms that can be understood by both software engineers and usability specialists. Usability specialists determine which aspects of usability are appropriate for a given task or application. Software engineers evaluate these aspects of usability within the context of the architecture and implement them within time, cost, performance, availability, security, and other constraints (Figure 1).

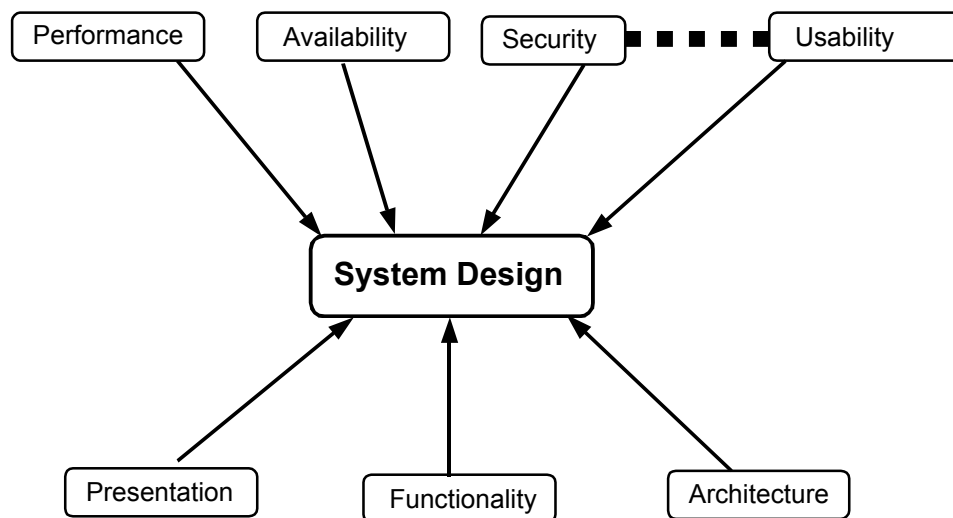


Figure 1. Usability is One Attribute of System Design Among Many

Architecture is one aspect of a system to be designed, as well as the presentation of information and the functionality of the system. Achieving better usability through software architecture is not a new goal. For the last twenty years, both researchers and practitioners have been concerned with the software architecture design techniques required to support usability. These techniques have focused on selecting the correct overall system structure, then showing how this structure will deliver the modifiability needed, while still supporting performance and other functionality. A review of these techniques can be found in Chapter 6 of *Software Architecture in Practice* [Bass 1998].

The architectures of the 1980s and early 1990s assumed that usability was primarily a property of the presentation of information. Therefore, simply separating the presentation from the dialog and the application made it easy to modify that presentation after user testing. However, that assumption proved insufficient to achieve usable systems. A more popular belief in the 1990s was that usability concerns greatly affected system functionality (application) as well as presentation. This new emphasis took away attention from architectural support (beyond separation of presentation and application). Achieving the correct functionality for a given system became paramount.

It is our observation that even if system presentation and functionality are designed extremely well, system usability can be greatly compromised if the underlying architecture does not support human concerns beyond modifiability. Many modifications only come to the fore after an initial design and implementation. As is well known to software engineers, the later in the life cycle that a problem is detected, the more expensive it is to fix. Furthermore, the architectural separation of presentation from application is insufficient to achieve usability because many usability concerns reach deep into the application, beyond the presentation layer. Cost and schedule pressures, then, often prevent many modifications from being implemented.

In this report, we take a proactive approach. Specifically, we present a series of connections between specific aspects of usability, such as the ability for a user to “undo,” and software architecture. Our contribution is to couple specific aspects of usability and architecture, rather than attempting to come up with the software architecture that satisfies all aspects of usability. Designers can use this information both to generate solutions to those aspects of usability required for their systems and to evaluate their systems for specific aspects of usability.

In addition to identifying the connections between aspects of usability and software architecture, we identify “general scenarios” (Section 2) that define those architecturally sensitive aspects of usability [Bass 2000]. We also categorize these scenarios into a hierarchy of human benefits (Sections 3 and 4) so that the design teams or stakeholders can evaluate their applicability to the system being constructed. We take a similar approach to architectural patterns. We provide a hierarchy of architectural mechanisms and then, for each scenario, we provide an architecture pattern that implements the scenario. We categorize these patterns according to their appropriate architecture mechanisms (Sections 5 and 6). Finally, we position the scenarios within a matrix of benefits and architecture mechanisms (Section 7).

The architecture patterns we provide will enable usability specialists to evaluate the impact of the proposed solution on other quality attributes, such as performance, availability, and security. In this way, we hope to give these specialists the tools necessary to decide which aspects of usability should be included in the beginning of the design process. We also hope to give software engineers the tools necessary to understand particular aspects of usability and their impact on total system quality.

Caveats: Collaboration and Ubiquitous Computing

Thus far, we have only applied usability benefit categories for individuals performing tasks on a desktop computer. However, increasing numbers of systems are being built to support collaborative groups. As a result, the benefits and general scenarios we describe need to be re-examined within the context of collaborative applications. In addition, increasing numbers of systems, such as laptops, personal digital assistants (PDAs), wearable computers, whiteboards, and other devices are being built to support ubiquitous computing. As a result, the benefits and general scenarios that we generated need to be reconsidered in non-desktop systems and applications. New general scenarios or benefits that arise in this context also need to be generated and cataloged. Suggestions are most welcome. This work is being done in an attempt to understand architectural mechanisms and to categorize software quality attributes. While it represents our thinking at this point, our ideas may evolve over time.

2 General Usability Scenarios

This section enumerates the usability scenarios that we have identified as being architecturally sensitive. A general usability scenario describes an interaction that some stakeholder (e.g., end user, developer, system administrator) has with the system under consideration from a usability point of view.

We generated the list of usability scenarios by surveying the literature, by personal experience, and by asking colleagues [Gram 1996, Newman 1995, Nielsen 1993]. We also screened the list so that all entries have explicit software architectural implications and solutions. Section 5 provides an architectural pattern that implements each scenario given in this report.

2.1 Aggregating Data

A user may want to perform one or more actions on more than one object. For example, an Adobe® Illustrator® user may want to enlarge many lines in a drawing. It could become tedious to perform these actions one at a time. Furthermore, the specific aggregations of actions or data that a user wishes to perform cannot be predicted; they result from the requirements of each task. Systems, therefore, should allow users to select and act upon arbitrary combinations of data.

2.2 Aggregating Commands

A user may want to complete a long-running, multi-step procedure consisting of several commands. For example, a psychology researcher may wish to execute a batch of commands on a data file during analysis. It could become tedious to invoke these commands one at a time, or to provide parameters for each command as it executes. If the computer is unable to accept the required inputs for this procedure up front, the user will be forced to wait for each input to be requested. Systems should provide a batch or macro capability to allow users to aggregate commands.

2.3 Canceling Commands

A user invokes an operation, then no longer wants the operation to be performed. The user now wants to stop the operation rather than wait for it to complete. It does not matter why the user launched the operation. The mouse could have slipped. The user could have mistaken one command for another. The user could have decided to invoke another operation. For these reasons (and many more), systems should allow users to cancel operations.

2.4 Using Applications Concurrently

A user may want to work with arbitrary combinations of applications concurrently. These applications may interfere with each other. For example, some versions of IBM® ViaVoice and Microsoft® Word contend for control of the cursor with unpredictable results. Systems should ensure that users can employ multiple applications concurrently without conflict.

(See: Supporting Multiple Activities)

2.5 Checking for Correctness

A user may make an error that he or she does not notice. However, human error is frequently circumscribed by the structure of the system; the nature of the task at hand, and by predictable perceptual, cognitive, and motor limitations. For example, users often type “hte” instead of “the” in word processors. The frequency of the word “the” in English and the fact that “hte” is not an English word, combined with the frequency of typing errors that involve switching letters typed by alternate hands, make automatically correcting to “the” almost always appropriate. Computer-aided correction becomes both possible and appropriate under such circumstances. Depending on context, error correction can be enforced directly (e.g., automatic text replacement, fields that only accept numbers) or suggested through system prompts.

2.6 Maintaining Device Independence

A user attempts to install a new device. The device may conflict with other devices already present in the system. Alternatively, the device may not function in certain specific applications. For example, a microphone that uses the Universal Serial Bus (USB) may fail to function with older sound software. Systems should be designed to reduce the severity and frequency of device conflicts. When device conflicts occur, the system should provide the information necessary to either solve the problem or seek assistance. (Devices include printers, storage/media, and I/O apparatus.)

2.7 Evaluating the System

A system designer or administrator may be unable to test a system for robustness, correctness, or usability in a systematic fashion. For example, the usability expert on a development team might want to log test users’ keystrokes, but may not have the facilities to do so. Systems should include test points and data gathering capabilities to facilitate evaluation.

2.8 Recovering from Failure

A system may suddenly stop functioning while a user is working. Such failures might include a loss of network connectivity or hard drive failure in a user’s PC. In these or other cases,

valuable data or effort may be lost. Users should be provided with the means to reduce the amount of work lost from system failures.

2.9 Retrieving Forgotten Passwords

A user may forget a password. Retrieving and/or changing it may be difficult or may cause lapses in security. Systems should provide alternative, secure mechanisms to grant users access. For example, some online stores ask each user for a maiden name, birthday, or the name of a favorite pet in lieu of a forgotten password.

2.10 Providing Good Help

A user needs help. The user may find, however, that a system's help procedures do not adapt adequately to the context. For example, a user's computer may crash. After rebooting, the help system automatically opens to a general table of contents rather than to a section on restoring lost data or searching for conflicts. Help content may also lack the depth of information required to address the user's problem. For example, an operating system's help area may contain an entry on customizing the desktop with an image, but may fail to provide a list of the types of image files that can be used. Help procedures should be context dependent and sufficiently complete to assist users in solving problems.

2.11 Reusing Information

A user may wish to move data from one part of a system to another. For example, a telemarketer may wish to move a large list of phone numbers from a word processor to a database. Re-entering this data by hand could be tedious and/or excessively time-consuming. Users should be provided with automatic (e.g., data propagation) or manual (e.g., cut and paste) data transports between different parts of a system. When such transports are available and easy to use, the user's ability to gain insight through multiple perspectives and/or analysis techniques will be enhanced.

2.12 Supporting International Use

A user may want to configure an application to communicate in his or her language or according to the norms of his or her culture. For example, a Japanese user may wish to configure the operating system to support a different keyboard layout. However, an application developed in one culture may contain elements that are confusing, offensive, or otherwise inappropriate in another. Systems should be easily configurable for deployment in multiple cultures.

2.13 Leveraging Human Knowledge

People use what they already know when approaching new situations. Such situations may include using new applications on a familiar platform, a new version of a familiar application, or a new product in an established product line.

New approaches usually bring new functionality or power. When, however, users are unable to apply what they already know, a corresponding cost in productivity and training time is incurred. For example, new versions of applications often assign items to different menus or change their names. As a result, users skilled in the older version are reduced to the level of novices again, searching menus for the function they know exists.

System designers should strive to develop upgrades that leverage users' knowledge of prior systems and allow them to move quickly and efficiently to the new system.

2.14 Modifying Interfaces

Iterative design is the lifeblood of current software development practice, yet a system developer may find it prohibitively difficult to change the user interface of an application to reflect new functions and/or new presentation desires. System designers should ensure that their user interfaces can be easily modified.

2.15 Supporting Multiple Activities

Users often need to work on multiple tasks more or less simultaneously (e.g., check mail and write a paper). A system or its applications should allow the user to switch quickly back and forth between these tasks.

2.16 Navigating Within a Single View

A user may want to navigate from data visible on-screen to data not currently displayed. For example, he or she may wish to jump from the letter "A" to the letter "Q" in an online encyclopedia without consulting the table of contents. If the system takes too long to display the new data or if the user must execute a cumbersome command sequence to arrive at her or his destination, the user's time will be wasted. System designers should strive to ensure that users can navigate within a view easily and attempt to keep wait times reasonably short.

(See: Working at the User's Pace)

2.17 Observing System State

A user may not be presented with the system state data necessary to operate the system (e.g., uninformative error messages, no file size given for folders). Alternatively, the system state may be presented in a way that violates human tolerances (e.g., is presented too quickly for people to read. See: Working at the User's Pace). The system state may also be presented in an unclear fashion, thereby confusing the user. System designers should account for human needs and capabilities when deciding what aspects of system state to display and how to present them.

A special case of Observing System State occurs when a user is unable to determine the level of security for data entered into a system. Such experiences may make the user hesitate to use the system or avoid it altogether.

2.18 Working at the User's Pace

A system might not accommodate a user's pace in performing an operation. This may make the user feel hurried or frustrated. For example, ATMs often beep incessantly when a user "fails" to insert an envelope in time. Also, Microsoft Word's scrolling algorithm does not take system speed into account and becomes unusable on fast systems (the data flies by too quickly for human comfort). Systems should account for human needs and capabilities when pacing the stages in an interaction. Systems should also allow users to adjust this pace as needed.

2.19 Predicting Task Duration

A user may want to work on another task while a system completes a long running operation. For example, an animator may want to leave the office to make copies or to eat while a computer renders frames. If systems do not provide expected task durations, users will be unable to make informed decisions about what to do while the computer "works." Thus, systems should present expected task durations.

2.20 Supporting Comprehensive Searching

A user wants to search some files or some aspects of those files for various types of content. For example, a user may wish to search text for a specific string or all movies for a particular frame. Search capabilities may be inconsistent across different systems and media, thereby limiting the user's opportunity to work. Systems should allow users to search data in a comprehensive and consistent manner by relevant criteria.

2.21 Supporting Undo

A user performs an operation, then no longer wants the effect of that operation. For example, a user may accidentally delete a paragraph in a document and wish to restore it. The system should allow the user to return to the state before that operation was performed. Furthermore, it is desirable that the user then be able to undo the prior operation (multi-level undo).

2.22 Working in an Unfamiliar Context

A user needs to work on a problem in a different context. Discrepancies between this new context and the one the user is accustomed to may interfere with the ability to work. For example, a clerk in business office A wants to post a payment for a customer of business unit B. Each business unit has a unique user interface, and the clerk has only used unit A's previously. The clerk may have trouble adapting to business unit B's interface (same system, unfamiliar context.) Systems should provide a novice (verbose) interface to offer guidance to users operating in unfamiliar contexts.

2.23 Verifying Resources

An application may fail to verify that necessary resources exist before beginning an operation. This failure may cause errors to occur unexpectedly during execution. For example, some versions of Adobe® PhotoShop® may begin to save a file only to run out of disk space before completing the operation. Applications should verify that all necessary resources are available before beginning an operation.

2.24 Operating Consistently Across Views

A user may become confused by functional deviations between different views of the same data. Commands that had been available in one view may become unavailable in another or may require different access methods. For example, users cannot run a spell check in the Outline View utility found in a mid-90's version of Microsoft Word. Systems should make commands available based on the type and content of a user's data, rather than the current view of that data, as long as those operations make sense in the current view.

For example, allowing users to perform operations on individual points in a scatter plot while viewing the plot at such a magnification that individual points cannot be visually distinguished does not make sense. A naïve user is likely to destroy the underlying data. The system should prevent selection of single points when their density exceeds the resolution of the screen, and inform the user how to zoom in, access the data in a more detailed view, or otherwise act on single data points.

(See: Providing Good Help and Supporting Visualization)

2.25 Making Views Accessible

Users often want to see data from other viewpoints. For example, a user may wish to see the outline of a long document and the details of the prose. If certain views become unavailable in certain modes of operation, or if switching between views is cumbersome, the user's ability to gain insight through multiple perspectives will be constrained.

(See: Supporting Visualization)

2.26 Supporting Visualization

A user wishes to see data from a different viewpoint. Systems should provide a reasonable set of task-related views to enhance users' ability to gain additional insight while solving problems. For example, Microsoft Word provides several views to help users compose documents, including Outline and Page Layout modes.

3 Details of the Usability Benefit Hierarchy

To create usable systems, designers must first ensure that their proposed products provide the functionality their users actually need to perform work as opposed to the functionality that the marketing or development team imagines they need. In other words, systems must provide functionality that fits the individual, organizational, and social structure of the work context. Although specifying and identifying needed functionality are fundamental steps in the development process, these design phases do not typically involve architectural concerns. Thus, we will not discuss them here. (We refer readers interested in these issues to *Contextual Design* [Beyer 1998].)

Assuming that the functionality needed by a system's users is correctly identified and specified, the usability of such a system can still be seriously compromised by architectural decisions that hinder or even prevent the required benefits. In extreme cases, the resulting system can become virtually unusable.

This section organizes and presents scenarios by their usability benefits. We arrived at the hierarchy of usability benefits presented in Table 1 using a bottom-up process called the affinity process [Beyer 1998]. We took this approach rather than taking an existing definition of usability and sorting the scenarios into it because it was not clear that architecturally sensitive scenarios would cover the typical range of usability benefits. However, the resulting hierarchy does not differ significantly from organizations of usability given by other authors [e.g., Newman 1995; Nielsen 1993; Shneiderman 1998], and we view this as partial confirmation that our set of architecturally sensitive scenarios covers, in some sense, the usability space. Each scenario occurs in one or more positions in the hierarchy.

The entries in this chapter discuss each item of the usability benefit hierarchy. One premise of this work has been that the design of a system embodies tradeoffs between benefits (usability) and cost (software engineering). Hence in each section, we discuss the appropriate messages for each benefit. This will enable the usability engineer to better argue the potential benefits of each scenario and the software engineer to know what instrumentation should be embedded into the system to support the benefit calculations.

Table 1. Usability Benefits Heirarchy

Increases individual user effectiveness

Expedites routine performance

Accelerates error-free portion of routine performance

Reduces the impact of routine user errors (slips)

Improves non-routine performance

Supports problem-solving

Facilitates learning

Reduces the impact of user errors caused by lack of knowledge (mistakes)

Prevents mistakes

Accommodates mistakes

Reduces the impact of system errors

Prevents system errors

Tolerates system errors

Increases user confidence and comfort

3.1 Increases Individual User Effectiveness

If addressed properly, the scenarios included in this category will improve the performance of individual users. Such increases in productivity, though seemingly small when considered discretely, can aggregate to produce substantial benefits for an organization as a whole.

3.1.1 Expedites routine performance

In a routine task, a user recognizes a situation, knows what the next goal should be, and knows what to do to accomplish that goal. No problem-solving is necessary. All that remains is for the user to recall and execute the commands necessary to complete the task.

When performing routine tasks, even skilled users will become faster but will probably not develop new methods to complete their tasks [Card 1983]. This is in contrast to a problem-solving or learning situation where the user is likely to discover or learn a new method while performing a task. (For an example of learning and problem-solving behavior, see non-routine performance.)

Although users know what to do to accomplish routine tasks, they will still make errors. In fact, observations of skilled users performing routine tasks reveal that about 20% of a user's time may be consumed by making, then recovering from, mistakes. These "routine errors" result from "slips" in execution (e.g., hitting the wrong key or selecting the menu item next to the one desired), rather than from a lack of knowledge (i.e., not knowing which command to

use). Slips can never be totally prevented if there are multiple actions available to a user, but some system designs accommodate these errors more successfully than others.

Accelerates error-free portion of routine performance

Routine tasks take time for a user to recognize the situation, recall the next goal and the method used to accomplish it, and to mentally and/or physically execute the commands to accomplish the goal. We call the minimum required time to accomplish a task, assuming no slips, the error-free portion of routine performance.

In practice, the actual performance time is the sum of this minimum time and the time it takes to make and recover from slips. Systems can be designed to maximize error-free performance time, thereby reducing time to perform routine tasks and increasing individual effectiveness.

Reduces the impact of routine user errors (slips)

The negative impact of routine user errors can be reduced in two ways. First, since users will always slip, reducing the number of opportunities for error (roughly corresponding to the number and difficulty of steps in a given procedure) will usually reduce its occurrence. Second, systems can be designed to better accommodate user slips by providing adequate recovery methods.

3.1.2 Improves non-routine performance

In a non-routine task, a user does not know exactly what to do. In this situation, the user may experiment within the interface by clicking on buttons either randomly or systematically to observe the effects. The user might guess at actions based on previous experience. He or she might also use a tutorial, a help system, or documentation. Success in these “weak methods” of dealing with a new situation can be helped or hindered through system design.

Supports problem-solving

Users employ problem-solving behavior when they do not know exactly what to do. This behavior can be described as a search through a problem space [Newell and Simon 1972]. When confronted with a new problem, people guess at solutions based on previous experience, try things at random to see what happens, or search for the desired effect.

For this discussion, we assume that the user understands the goal of the task (e.g., I would like to replace all occurrences of “bush” with “shrub”), but the user may have to search through the system’s available commands to achieve the desired outcome.

Measures of how well a system supports problem-solving include

- the time it takes to accomplish a novel task

- the number of incorrect paths the user takes while accomplishing a novel task
- the type of incorrect paths the user takes while accomplishing a novel task (e.g., paths that have unforeseen and permanent side effects or benign paths that change nothing but simply add to the problem-solving time)
- the time necessary to recover from incorrect paths (Systems that support UNDO usually score well on this measure.)

In addition to reducing time spent on incorrect paths, well-designed systems may actually enhance users' problem-solving capabilities, further improving productivity.

Facilitates learning

Humans continuously learn as they perform tasks. Even in routine situations, humans continue to speed up with each repetition, eventually reaching a plateau where further improvements in performance become nearly imperceptible. In non-routine situations, people learn by receiving training, consulting instructions (using a help system, documentation, or asking a friend), by exploring the system, by applying previous experience to the new situation, and/or by reasoning based on what they know (or think they know) about a system. They may also learn by making a mistake, observing that the erroneous action does not produce the desired result, and by remembering not to perform this action again.

Measures of how well a system supports learning typically include

- the number of times a task must be performed by a user before it is completed without error. (Often investigators include a repetition requirement to avoid the "luck" factor; for example, a user must perform a task n times without error.)
- the time before a user fulfills the error-free repetition requirement (defined above)
- incidental learning measures, in which a user first performs a task until some level of mastery is reached. The user then performs a different task that he or she has not practiced. The problem-solving and learning measures associated with this second task are measures of incidental learning.

3.1.3 Reduces the impact of user errors caused by lack of knowledge (mistakes)

In addition to the errors people make even when they know how to accomplish their tasks (slips, discussed above), people make errors when they do not know what to do in the current situation. In a typical scenario, a user does not understand that the current situation differs in important ways from previously encountered situations, and therefore he or she misapplies

knowledge of procedures that have worked before.¹ Errors due to lack of knowledge are called mistakes.

Design cannot prevent all mistakes, but careful design can prevent some of them. For example, a typical technique to help prevent mistakes is to gray out inapplicable menu items. Since some mistakes will still occur, systems should also be designed to accommodate them.

Prevents mistakes

The following are typical measures of how well a system helps to prevent mistakes:

- the number of mistaken actions that a user could make while completing a task
- the type of mistakes the user could make while accomplishing a task (e.g., paths that have unforeseen and permanent side effects, or benign paths that change nothing)

(While these measures appear similar to those associated with problem-solving; that case focuses on how well the system guides the user back to the correct path. Preventing mistakes focuses on how well the system guides the user away from an incorrect path. The difference is subtle.)

Accommodates mistakes

Since mistakes will occur if the user has the freedom to stray from a correct path, the system should accommodate these errors. The most telling measures of such accommodation are

- the degree to which the system can be restored to the state prior to the mistake
- the time necessary to recover from mistakes (Systems that support UNDO usually score well on this measure.) This duration includes the time needed to restore all data and resources to the state before the error.

3.2 Reduces the Impact of System Errors

Systems will always operate with some degree of error. Networks will go down, power failures will occur, and applications will contend for resources and conflict. Design cannot prevent all system errors, but careful design can prevent some of them. All systems should be designed to tolerate system errors. This section differs from section 3.1. “Reduces the impact of routine user errors” only in the source of the error discussed. Here, we address system

¹ It is often difficult to distinguish a mistake from an exploratory problem-solving action. Typically, a mistake is when the user “knows” what to do and is wrong; while problem-solving is when the user doesn’t know what to do and is trying to find the correct way. Therefore, the difference can only be detected through means other than the observation of actions – think-aloud protocols or interviews about what a person intended when taking an action, or his or her response when the action does not have the intended result (which indicates a mistake) typically allow observers to make this distinction. However, for architecture design, this distinction is not important; some users may be problem-solving and others making mistakes, but the architecture should support both.

error, not user error. The measures stay the same but the object of measurement becomes the system.

3.2.1 Prevents system errors

As with preventing mistakes, the measures associated with preventing system errors are the number and type of error that occur as a user performs a task.

3.2.2 Tolerates system errors

Since system errors *will* occur, systems should be set up to tolerate them. Again, as with accommodating mistakes, the most telling measures of error tolerance are

- the degree to which the system state can be restored to the state before the error.
- the time necessary to recover from errors. This duration includes the time needed to restore all data and resources to the system state before the error.

3.3 Increases user confidence and comfort

In the scenarios included in this category, the benefits do not involve users' efficiency, problem-solving processes, ability to learn, or propensity to make mistakes. The benefits do involve how they feel about the system; for some architectural decisions do facilitate or inhibit capabilities that increase user confidence and comfort, and this may be of value to an organization. Measures of confidence and comfort are more indirect than the time- and error-based metrics in the preceding categories, and typically involve questionnaires or interviews, or analysis of buying behavior (e.g., return customers and referrals).

4 Categorization By Usability Benefit

As stated in Section 3, we organized the general usability scenarios into a hierarchy by benefit to the user. The hierarchy that results was presented in Table 1, is shown again as Table 2.

Table 2. Usability Benefits Heirarchy

Increases individual user effectiveness

Expedites routine performance

Accelerates error-free portion of routine performance

Reduces the impact of routine user errors (slips)

Improves non-routine performance

Supports problem-solving

Facilitates learning

Reduces the impact of user errors caused by lack of knowledge (mistakes)

Prevents mistakes

Accommodates mistakes

Reduces the impact of system errors

Prevents system errors

Tolerates system errors

Increases user confidence and comfort

This hierarchy is of benefit as a screening mechanism for the scenario. That is, if you are a system designer or evaluator, you can characterize the usability requirements for your system in terms of the categories in the hierarchy. The scenarios that appear are those that may be necessary to achieve your requirements.

We now enumerate the scenarios again and justify placing the scenarios in particular categories. A single scenario may appear in multiple categories. For example, undo supports routine performance (correcting slips) but it also supports exploration (the user can always recover from a test path).

The form of each section is the same. We repeat the scenario and present its entry in the benefit hierarchy. We then provide a brief argument that justifies the scenarios placement in the hierarchy.

4.1 Aggregating Data

A user may want to perform one or more actions on more than one object. For example, an Adobe® Illustrator® user may want to enlarge many lines in a drawing. It could become tedious to perform these actions one at a time. Furthermore, the specific aggregations of actions or data that a user wishes to perform cannot be predicted; they result from the requirements of each task. Systems, therefore, should allow users to select and act upon arbitrary combinations of data.

Allocation to Benefit Hierarchy

Increases individual user effectiveness

Expedites routine performance

Accelerates error-free portion of routine performance

Data aggregation makes the system rather than the user responsible for iteration. This often saves time, thereby accelerating routing performance [Bhavani 2000].

Increases individual user effectiveness

Reduces the impact of user errors caused by lack of knowledge (mistakes)

Prevents mistakes

Operating on aggregates usually requires fewer human actions (e.g., typing, mouse movements) than working on each member of the aggregate in turn. Since performing more actions introduces more opportunities for error, the presence of data aggregation functionality can prevent slips [Miller 1987]. Some opportunities for error still exist, however, and errors committed on aggregates can be “larger” in a sense. For instance, if you change a Microsoft Word style to *italic* when you intended to enter **bold**, you have changed text throughout the document instead of only in one place. Still, aggregation functionality may facilitate recovery from such mistakes, so error analysis might be useful to understand the tradeoffs particular to a given system. Generally, the benefits of supporting aggregation far outweigh the risks.

Increases individual user effectiveness

Reduces the impact of user errors caused by lack of knowledge (mistakes)

Accommodates mistakes (Dependant on Search)

When working with aggregates, a robust search system can help accommodate user mistakes. In particular, a user may forget the members of an aggregate. To accommodate such errors, the user must be able to search by aggregate. Other mistakes can be accommodated by enabling the system to search by the criteria that users employed to create aggregates (type, modification date, etc.).

4.2 Aggregating Commands

A user may want to complete a long-running, multi-step procedure consisting of several commands. For example, a psychology researcher may wish to execute a batch of commands

a data file during analysis. It could become tedious to invoke these commands one at a time, or to provide parameters for each command as it executes. If the computer is unable to accept the required inputs for this procedure up front, the user will be forced to wait for each input to be requested. Systems should provide a batch or macro capability to allow users to aggregate commands.

Allocation to Benefit Hierarchy

Increases individual effectiveness

Expedites routine performance

Accelerates error-free portion

The ability to aggregate commands enhances routine performance by allowing users to work on multiple tasks at the same time. For example, a user can get coffee while the computer prints a queue of documents. (This is an aggregation of individual print commands.)

Increases individual effectiveness

Expedites routine performance

Reduces impact of slips

Executing a macro, batch, or script requires fewer human actions (e.g., typing, mouse movements) than executing commands in turn. Since performing more actions introduces more opportunities for error, the presence of command aggregation functionality can prevent slips.

Increases individual effectiveness

Reduces the impact of user errors caused by lack of knowledge (mistakes)

Prevents mistakes

Executing a macro, batch, or script requires little thought and presents fewer opportunities for user mistakes than applying commands one by one in sequence. Thus, once the macro or script is written, tested, and debugged, subsequent use of that macro or script will be virtually error-free. Analogous to the problem with aggregation of data, there is the danger of producing an incorrect macro or script that will have far reaching consequences, but the potential for benefit provided by this capability usually far outweighs the potential for damage.

Increases individual effectiveness

Reduces the impact of user errors caused by lack of knowledge (mistakes)

Accommodates mistakes (Dependent on Search)

When working with command aggregates, a robust search system can help accommodate user mistakes. In particular, a user may forget the name of the macro or script. The forgetting of the name of the macro or script is a trade-off with forgetting the elements included in the macro or script. To accommodate the forgetting of the name of the macro or script, the user must be able to search by whatever criteria users might employ (content, type of file it works on, modification date, etc.) to create command aggregates.

4.3 Canceling Commands

A user invokes an operation, then no longer wants the operation to be performed. The user now wants to stop the operation rather than wait for it to complete. It does not matter why the user launched the operation. The mouse could have slipped. The user could have mistaken one command for another. The user could have decided to invoke another operation. For these reasons (and many more), systems should allow users to cancel operations.

Allocation to Benefit Hierarchy

Increases individual effectiveness

Expedites routine performance

Reduces impact of slips

Cancellation reduces the impact of slips by allowing users to revoke accidental commands.

Increases individual effectiveness

Improves non-routine performance

Supports problem-solving

Cancellation facilitates problem-solving by allowing users to apply commands and explore without fear, because they can always abort their actions.

Increases individual effectiveness

Reduces the impact of user errors caused by lack of knowledge (mistakes)

Accommodates mistakes

Cancellation accommodates user mistakes by allowing users to abort commands they invoke through lack of knowledge.

Reduces the impact of system errors

Tolerates system errors

Cancellation helps users tolerate system error by allowing users to abort commands that aren't working properly (for example, a user cancels a download because the network is jammed).

4.4 Using Applications Concurrently

A user may want to work with arbitrary combinations of applications concurrently. These applications may interfere with each other. For example, some versions of IBM® ViaVoice and Microsoft® Word contend for control of the cursor with unpredictable results. Systems should ensure that users can employ multiple applications concurrently without conflict.

(See: Supporting Multiple Activities)

Allocation to Benefit Hierarchy

Increases individual effectiveness

Expedites routine performance

Accelerates error-free portion

When applications interfere with each other, they disrupt routine performance. Also, when systems are designed to support concurrent applications, users may switch more easily from task to task or work on several tasks at the same time.

Increases individual effectiveness

Improves non-routine performance

Supports problem-solving

Interfering applications may inhibit a user's ability to solve problems. A user may take an action while exploring a path that has an unintended and undesirable result because of an application interaction, rather than because the action itself was incorrect. This may cause the user to abandon paths of exploration that would have led to solutions.

Increases individual effectiveness

Improves non-routine performance

Facilitates learning

When applications interfere with each other, users may learn superstitious behavior. For example, a user may think that a system crash was caused by a personal mistake, when in fact two applications interfered with each other.

Increases individual effectiveness

Reduces the impact of user errors caused by lack of knowledge (mistakes)

Prevents mistakes

Application interference is an instance of a system error. Systems that are designed to support concurrent applications prevent this problem.

4.5 Checking for Correctness

A user may make an error that he or she does not notice. However, human error is frequently circumscribed by the structure of the system, the nature of the task at hand, and by predictable perceptual, cognitive, and motor limitations. For example, users often type "hte" instead of "the" in word processors. The frequency of the word "the" in English and the fact that "hte" is not an English word, combined with the frequency of typing errors that involve switching letters typed by alternate hands, make automatically correcting to "the" almost always appropriate. Computer-aided correction becomes both possible and appropriate under such circumstances. Depending on context, error correction can be enforced directly (e.g., automatic text replacement, fields that only accept numbers) or suggested through system prompts.

Allocation to Benefit Hierarchy

Increases individual effectiveness

Expedites routine performance

Reduces impact of slips

When systems are designed to correct very predictable slips (e.g., typing “hte” instead of “the”), the effects of user slips can be reduced.

Increases individual effectiveness

Improves non-routine performance

Supports problem-solving

Since undetected human errors can lead users down paths in the problem space that have nothing to do with solutions, correctness checking can help keep users on a more context-appropriate and direct path when problem-solving.

Increases individual effectiveness

Improves non-routine performance

Facilitates learning

As described above, undetected human errors can lead users down paths in the problem space that have nothing to do with solutions. Correctness checking can help keep users on a more context appropriate and direct path when problem solving. Direct paths are easier to learn. Furthermore, a system that corrects errors and informs users of their mistakes can facilitate learning by providing additional feedback, and sometimes even the correct answer. For example, a person can learn the correct spelling of a word from a spell checker.

Increases individual effectiveness

Reduces the impact of user errors caused by lack of knowledge (mistakes)

Prevents mistakes

When systems are designed to correct errors, user mistakes can be prevented. For example, a system could ensure that only certain numerals are entered into the ID search field of a document.

Increases individual effectiveness

Reduces the impact of user errors caused by lack of knowledge (mistakes)

Accommodates mistakes

As with slips, when systems are designed to correct errors, the effects of user mistakes can be reduced. Finding and fixing mistakes also can be much easier. For instance, a user can spell a word by guessing, use the automatically generated suggestions of a spell-checker to find the correct spelling, and insert it.

4.6 Maintaining Device Independence

A user attempts to install a new device. The device may conflict with other devices already present in the system. Alternatively, the device may not function in certain specific applications. For example, a microphone that uses the Universal Serial Bus (USB) may fail to function with older sound software. Systems should be designed to reduce the severity and frequency of device conflicts. When device conflicts occur, the system should provide the information necessary to either solve the problem or seek assistance. (Devices include printers, storage/media, and I/O apparatus.)

Allocation to Benefit Hierarchy

Increases individual effectiveness

Expedites routine performance

Accelerates error-free portion of routine performance

If the user swaps peripherals frequently, (for example, swapping a CD-ROM drive for a 3.5" drive) then device independence will accelerate routine performance.

Increases individual effectiveness

Improves non-routine performance

Supports problem-solving

If a user encounters an unfamiliar or undesirable device in an occasional situation, device independence will allow the user easily to switch to a more familiar or desirable device. For example, a user with Repetitive Stress Injury may have a special keyboard. If it can be transported and installed to any system easily, the user will be able to type anywhere.

Reduces impact of system errors

Prevents system errors

Device errors are system errors. Systems that are designed to support multiple, independent devices prevent these errors from occurring.

4.7 Evaluating the System

A system designer or administrator may be unable to test a system for robustness, correctness, or usability in a systematic fashion. For example, the usability expert on a development team might want to log test users' keystrokes, but may not have the facilities to do so. Systems should include test points and data gathering capabilities to facilitate evaluation.

Allocation to Benefit Hierarchy

Increases individual effectiveness

Expedites routine performance

Accelerates error-free portion of routine performance

Testing is a routine part of programming. Systems that are easy to evaluate are easier to test. This, in turn, can accelerate the routine performance of software developers.

Increases individual effectiveness

Improves non-routine performance

Supports problem solving

When testing uncovers a bug, software developers must problem-solve. Systems that are easy to evaluate enable programmers quickly to see the results of the changes they make at various checkpoints.

4.8 Recovering From Failure

A system may suddenly stop functioning while a user is working. Such failures might include a loss of network connectivity or hard drive failure in a user's PC. In these or other cases, valuable data or effort may be lost. Users should be provided with the means to reduce the amount of work lost from system failures.

Allocation to Benefit Hierarchy

Reduces impact of system errors

Tolerates system errors

System failure is the most extreme case of system error. Though systems that support failure recovery are no less prone to failure, they at least help users tolerate such mishaps by facilitating data restoration.

4.9 Retrieving Forgotten Passwords

A user may forget a password. Retrieving and/or changing it may be difficult or may cause lapses in security. Systems should provide alternative, secure mechanisms to grant users access. For example, some online stores ask each user for a maiden name, birthday, or the name of a favorite pet in lieu of a forgotten password.

Allocation to Benefit Hierarchy

Increases individual effectiveness

Reduces the impact of user errors caused by lack of knowledge (mistakes)

Accommodates mistakes

When users forget passwords, they lack the knowledge to access their accounts in the normal way. Providing alternative methods of entering protected sites, or means to retrieve forgotten passwords, helps to accommodate such errors.

4.10 Providing Good Help

A user needs help. The user may find, however, that a system's help procedures do not adapt adequately to the context. For example, a user's computer may crash. After rebooting, the help system automatically opens to a general table of contents rather than to a section on restoring lost data or searching for conflicts. Help content may also lack the depth of information required to address the user's problem. For example, an operating system's help area may contain an entry on customizing the desktop with an image, but may fail to provide a list of the types of image files that can be used. Help procedures should be context dependent and sufficiently complete to assist users in solving problems.

Allocation to Benefit Hierarchy

Increases individual effectiveness

Improves non-routine performance

Supports problem solving

Feedback is essential to human problem-solving. Thus, users may solve problems more effectively by seeking additional feedback from context dependent help systems. If the help is correct, understandable, and delivered in the right context, it can simply tell the user what to do to solve the problem.

Increases individual effectiveness

Improves non-routine performance

Facilitates learning

As with problem solving, feedback is an essential component of human learning processes. Users may learn more effectively by seeking additional feedback from context-dependent help systems. If the help is correct, understandable, and delivered in the right context, the user can learn the correct concept or procedure from the help material.

4.11 Reusing Information

A user may wish to move data from one part of a system to another. For example, a telemarketer may wish to move a large list of phone numbers from a word processor to a database. Re-entering this data by hand could be tedious and/or excessively time-consuming. Users should be provided with automatic (e.g., data propagation) or manual (e.g., cut and paste) data transports between different parts of a system. When such transports are available and easy to use, the user's ability to gain insight through multiple perspectives and/or analysis techniques will be enhanced.

Allocation to Benefit Hierarchy

Increases individual effectiveness

Expedites routine performance

Accelerates error-free portion of routine performance

In most cases, it is more efficient for systems to transport information from place to place than it is for users to re-enter this information by hand. Thus, systems that support information reuse accelerate routine performance.

Increases individual effectiveness

Expedites routine performance

Reduces impact of slips

Automatic data transportation and/or re-entry require fewer human actions (e.g., typing, mouse movements) than re-entering data by hand. Since performing more actions introduces more opportunities for error, systems that support information reuse can prevent slips.

Increases individual effectiveness

Improves non-routine performance

Supports problem-solving

When users can import and export data from one place to another easily, they may try different applications to gain additional insight while solving problems. For example, a user may export data from a traditional text-based statistics application to a data visualization application. Thus, systems that support information reuse facilitate problem-solving.

4.12 Supporting International Use

A user may want to configure an application to communicate in his or her language or according to the norms of his or her culture. For example, a Japanese user may wish to configure the operating system to support a different keyboard layout. However, an application developed in one culture may contain elements that are confusing, offensive, or otherwise inappropriate in another. Systems should be easily configurable for deployment in multiple cultures.

Allocation to Benefit Hierarchy

Increases individual effectiveness

Expedites routine performance

Accelerates error-free portion of routine performance

Systems that support internationalization accelerate users' performance by allowing them to communicate with the system in the language that they know best.

Increases individual effectiveness

Expedites routine performance

Reduces impact of slips

Systems that support internationalization help accommodate users' slips by presenting error messages in the language that they know best.

Increases individual effectiveness
Improves non-routine performance
Supports problem-solving

Systems that support internationalization facilitate problem-solving by allowing users to receive feedback from the system in the language that they know best.

Increases individual effectiveness
Improves non-routine performance
Facilitates learning

Systems that support internationalization facilitate learning by allowing users to receive feedback from the system in the language that they know best.

Increases individual effectiveness
Reduces the impact of user errors caused by lack of knowledge (mistakes)
Prevents mistakes

Systems that support internationalization help users avoid linguistic mistakes by allowing them to communicate with the system in the language that they know best.

Increases individual effectiveness
Reduces the impact of user errors caused by lack of knowledge (mistakes)
Accommodates mistakes

Systems that support internationalization help accommodate user mistakes by presenting error messages in the language that they know best. Incomprehensible error messages can compound existing misunderstanding.

Increases confidence and comfort

Being able to communicate with a system in the language that a user knows best reduces frustration and increases user satisfaction by affirming the importance of the user's national or cultural identity.

4.13 Leveraging Human Knowledge

People use what they already know when approaching new situations. Such situations may include using new applications on a familiar platform, a new version of a familiar application, or a new product in an established product line.

New approaches usually bring new functionality or power. When, however, users are unable to apply what they already know, a corresponding cost in productivity and training time is incurred. For example, new versions of applications often assign items to different menus or

change their names. As a result, users skilled in the older version are reduced to the level of novices again, searching menus for the function they know exists.

System designers should strive to develop upgrades that leverage users' knowledge of prior systems and allow them to move quickly and efficiently to the new system.

Allocation to Benefit Hierarchy

Increases individual effectiveness

Expedites routine performance

Accelerates error-free portion of routine performance

Leveraging human knowledge can directly benefit both software developers and users. For example, reusing the code for a search engine's interface leverages the end-user's knowledge of how to use the search engine. At the same time, it reduces the amount of new code that has to be written.

Increases individual effectiveness

Improves non-routine performance

Supports problem solving

When problem solving, users first apply the methods that they have used successfully in other parts of a system. By leveraging human knowledge, designers increase the likelihood that a user's early explorations will be effective.

Increases individual effectiveness

Improves non-routine performance

Facilitates learning

By leveraging human knowledge, designers reduce the amount of information that users must acquire to operate their systems. Thus, users can devote their time to learning more advanced techniques, techniques that match their interests. The quality of their learning is therefore increased.

Increases individual effectiveness

Reduces the impact of user errors caused by lack of knowledge (mistakes)

Prevents mistakes

If the system is designed so that old methods are recognized as legitimate (rather than as "errors"), leveraging human knowledge can prevent errors. It is not necessary for the new system to react the same way as the old system did. Rather, the system could be designed to do something sensible with old methods (for example., bring up context sensitive help that directs the user to the new way of accomplishing the goal). Thus, using old methods becomes a learning opportunity rather than an "error" from which the user must recover.

4.14 Modifying Interfaces

Iterative design is the lifeblood of current software development practice, yet a system developer may find it prohibitively difficult to change the user interface of an application to reflect new functions and/or new presentation desires. System designers should ensure that their user interfaces can be easily modified.

Allocation to Benefit Hierarchy

Increases individual effectiveness

Expedites routine performance

Accelerates error-free portion of routine performance

Developers must routinely modify the appearance and functionality of visual interfaces. Systems that reduce the overhead involved in making such changes accelerate the routine performance of programmers.

Increases individual effectiveness

Reduces the impact of routine user errors (slips)

Accommodates mistakes

The need to modify a user interface can arise from a lack of knowledge on the part of the development team regarding user needs and expectations. In such cases, systems that reduce the overhead involved in making interface changes help to reduce the impact of mistakes.

4.15 Supporting Multiple Activities

Users often need to work on multiple tasks more or less simultaneously (e.g., check mail and write a paper). A system or its applications should allow the user to switch quickly back and forth between these tasks.

Allocation to Benefit Hierarchy

Increases individual effectiveness

Expedites routine performance

Accelerates error-free portion of routine performance

During the course of routine work, users may want to tackle several tasks at once, or to open several applications concurrently to complete a single task. Users may also wish to work on a different task while waiting for the computer to complete an operation. For example, a user may want to read email while a Web page loads. Systems that support multiple activities facilitate and increase routine performance.

4.16 Navigating Within a Single View

A user may want to navigate from data visible on-screen to data not currently displayed. For example, he or she may wish to jump from the letter “A” to the letter “Q” in an online encyclopedia without consulting the table of contents. If the system takes too long to display the new data or if the user must execute a cumbersome command sequence to arrive at her or his destination, the user’s time will be wasted. System designers should strive to ensure that users can navigate within a view easily and attempt to keep wait times reasonably short.

(See: Working at the User’s Pace)

Allocation to Benefit Hierarchy

Increases individual effectiveness

Expedites routine performance

Accelerates error-free portion of routine performance

This is a ubiquitous routine activity in complex computer systems with data sets of any size (e.g., a paper more than a page long, a spreadsheet larger than the screen). Making such navigation possible, easy, and fast, speeds routine performance of almost any task.

4.17 Observing System State

A user may not be presented with the system state data necessary to operate the system (e.g., uninformative error messages, no file size given for folders). Alternatively, the system state may be presented in a way that violates human tolerances (e.g., is presented too quickly for people to read. See: Working at the User’s Pace). The system state may also be presented in an unclear fashion, thereby confusing the user. System designers should account for human needs and capabilities when deciding what aspects of system state to display and how to present them.

A special case of Observing System State occurs when a user is unable to determine the level of security for data entered into a system. Such experiences may make the user hesitate to use the system or avoid it altogether.

Allocation to Benefit Hierarchy

Increases individual effectiveness

Expedites routine performance

Reduces impact of slips

When the inevitable slip happens, if the system state is readily and easily observed, the user will know how to correct the slip before continuing further down an incorrect path.

Increases individual effectiveness

Improves non-routine performance
Supports problem solving

Human problem-solving depends on knowledge of current state (where you are), the goal state (where you want to be), and awareness of the range of available actions. Thus, being able to observe the current system state is central to the process of problem solving.

Increases individual effectiveness
Improves non-routine performance
Facilitates learning

Learning correct actions depends on knowing the system state when the action produced a desired response. Thus, if the system state is obscured or unobservable, the user's ability to learn will be inhibited.

Increases individual effectiveness
Reduces the impact of user errors caused by lack of knowledge (mistakes)
Prevents mistakes

A common type of mistake occurs when a user applies knowledge and procedures appropriate to one system state to a different, inappropriate, system state. Making the system state easily available to users reduces the likelihood of this type of mistake.

Increases individual effectiveness
Reduces the impact of user errors caused by lack of knowledge (mistakes)
Accommodates mistakes

If the system state is readily and easily observed, the user will know how to correct the mistake before continuing further down an incorrect path.

Increases confidence and comfort

This applies to the special case of the user being unable to determine the level of security for data entered into a system. Such experiences may make the user hesitate to use the system or avoid it altogether. Thus, systems that prominently display security policies and security levels (both of which are features of system state) increase user confidence and comfort.

4.18 Working at the User's Pace

A system might not accommodate a user's pace in performing an operation. This may make the user feel hurried or frustrated. For example, ATMs often beep incessantly when a user "fails" to insert an envelope in time. Also, Microsoft Word's scrolling algorithm does not take system speed into account and becomes unusable on fast systems (the data flies by too quickly for human comfort). Systems should account for human needs and capabilities when pacing the stages in an interaction. Systems should also allow users to adjust this pace as needed.

Allocation to Benefit Hierarchy

Increases individual effectiveness

Expedites routine performance

Accelerates error-free portion of routine performance

Systems that move either too slowly or too swiftly (as in the scrolling example above) interfere with routine performance. The user may actually change to a less-efficient method to work around the pace mismatch.

Increases individual effectiveness

Expedites routine performance

Reduces impact of slips

Systems that move too quickly may push users to, or past, their perceptual limits and increase the likelihood of slips. For example, scrollbars that do not take system speed into account may move too quickly on fast systems and cause users to make errors when navigating documents.

Increases confidence and comfort

Even in cases where errors are not introduced, systems that do not support pace tolerance may make users feel frustrated or hurried.

4.19 Predicting Task Duration

A user may want to work on another task while a system completes a long running operation. For example, an animator may want to leave the office to make copies or to eat while a computer renders frames. If systems do not provide expected task durations, users will be unable to make informed decisions about what to do while the computer “works.” Thus, systems should present expected task durations.

Allocation to Benefit Hierarchy

Increases individual effectiveness

Expedites routine performance

Accelerates error-free portion of routine performance

If performing long tasks is part of routine performance, allowing the user to do other things instead of waiting accelerates overall performance.

Increases individual effectiveness

Reduces the impact of user errors caused by lack of knowledge (mistakes)

Prevents mistakes

If the system accurately predicts the time it will take to accomplish an operation, the user will not assume that inactivity indicates that the system is “hung.” Thus, the user will not make the mistake of canceling the operation or rebooting the machine.

4.20 Supporting Comprehensive Searching

A user wants to search some files or some aspects of those files for various types of content. For example, a user may wish to search text for a specific string or all movies for a particular frame. Search capabilities may be inconsistent across different systems and media, thereby limiting the user’s opportunity to work. Systems should allow users to search data in a comprehensive and consistent manner by relevant criteria.

Allocation to Benefit Hierarchy

Increases individual effectiveness

Expedites routine performance

Accelerates error-free portion of routine performance

During routine work, using a search feature is often more efficient than perusing a large amount of material. Thus, systems that support search accelerate routine performance.

Increases individual effectiveness

Improves non-routine performance

Supports problem solving

People solve problems more easily if the path to success is straightforward. Systems that support search can facilitate problem-solving by helping users stay on track. For example, users often employ searches in online help systems to find relevant information.

Increases individual effectiveness

Improves non-routine performance

Facilitates learning

People learn more easily if the path to success is straightforward. Systems that support search can facilitate learning by helping users stay on track. (See example above.)

Increases individual effectiveness

Reduces the impact of user errors caused by lack of knowledge (mistakes)

Prevents mistakes

Automated searching is much more accurate than visually searching a lot of material. Therefore, supporting search prevents mistakes attributed to overlooking a target item.

4.21 Supporting Undo

A user performs an operation, then no longer wants the effect of that operation. For example, a user may accidentally delete a paragraph in a document and wish to restore it. The system

should allow the user to return to the state before that operation was performed. Furthermore, it is desirable that the user then be able to undo the prior operation (multi-level undo).

Allocation to Benefit Hierarchy

Increases individual effectiveness

- Expedites routine performance**
- Reduces impact of slips**

Undo reduces the impact of user slips by allowing users to return to the state prior to any accidental actions.

Increases individual effectiveness

- Improves non-routine performance**
- Supports problem solving**

Undo facilitates problem-solving by allowing users to apply commands and explore without fear, because they can always return to the previous state.

Increases individual effectiveness

- Reduces the impact of user errors caused by lack of knowledge (mistakes)**
- Accommodates mistakes**

Undo accommodates user mistakes by allowing users to return to the state prior to any accidental actions.

4.22 Working in an Unfamiliar Context

A user needs to work on a problem in a different context. Discrepancies between this new context and the accustomed one may interfere with the user's ability to work. For example, a clerk in business office A wants to post a payment for a customer of business unit B. Each business unit has a unique user interface, and the clerk has only used unit A's previously. The clerk may have trouble adapting to business unit B's interface (same system, unfamiliar context). Systems should provide a novice (verbose) interface to offer guidance to users operating in unfamiliar contexts.

Allocation to Benefit Hierarchy

Increases individual effectiveness

- Improves non-routine performance**
- Supports problem solving**

A novice or verbose interface can support problem-solving by giving the user the additional or enhanced feedback needed to consider various options.

Increases individual effectiveness

- Reduces the impact of user errors caused by lack of knowledge (mistakes)**

Prevents mistakes

A novice or verbose interface can give the user additional or enhanced feedback. This information will increase the user's knowledge and reduce the likelihood of mistakes.

4.23 Verifying Resources

An application may fail to verify that necessary resources exist before beginning an operation. This failure may cause errors to occur unexpectedly during execution. For example, some versions of Adobe® PhotoShop® may begin to save a file only to run out of disk space before completing the operation. Applications should verify that all necessary resources are available before beginning an operation.

Allocation to Benefit Hierarchy

Increases individual effectiveness

Expedites routine performance

Accelerates error-free portion of routine performance

Routine performance can be compromised if a system wastes the user's time by invoking a sequence of actions that cannot be completed. Systems that verify resources sidestep this problem.

Reduces impact of system errors

Prevents system errors

Some system errors result when one or more components or applications vie for a given resource. Systems that verify resources avoid this problem.

4.24 Operating Consistently Across Views

A user may become confused by functional deviations between different views of the same data. Commands that had been available in one view may become unavailable in another or may require different access methods. For example, users cannot run a spell check in the Outline View utility found in a mid-90's version of Microsoft Word. Systems should make commands available based on the type and content of a user's data, rather than the current view of that data, as long as those operations make sense in the current view.

For example, allowing users to perform operations on individual points in a scatter plot while viewing the plot at such a magnification that individual points cannot be visually distinguished does not make sense. A naïve user is likely to destroy the underlying data. The system should prevent selection of single points when their density exceeds the resolution of the screen, and inform the user how to zoom in, access the data in a more detailed view, or otherwise act on single data points.

(See: Providing Good Help and Supporting Visualization)

Allocation to Benefit Hierarchy

Increases individual effectiveness

Expedites routine performance

Accelerates error-free portion of routine performance

If all views in an application afford the same operations, routine performance will be enhanced; users will no longer need to navigate to a specific view to perform some action.

Increases individual effectiveness

Improves non-routine performance

Supports problem solving

When problem solving, users first apply the methods that they have used successfully in other parts of a system. By making all views operate consistently, designers increase the likelihood that a user's early explorations in a given view will be effective, thereby enhancing the user's ability to solve problems.

Increases individual effectiveness

Improves non-routine performance

Facilitates learning

By making all views operate consistently, designers reduce the amount of information that a user must learn to operate a given system.

Increases individual effectiveness

Reduces the impact of user errors caused by lack of knowledge (mistakes)

Prevents mistakes

When faced with a new or infrequently used view, users will attempt to apply the methods that have been successful with other parts of the system. When discrepancies between the operations afforded by different views are reduced, the likelihood that users will make such mis-application mistakes is also reduced.

4.25 Making Views Accessible

A user may want to see data from another point of view. For example, a user may wish to see the outline of a long document and the details of the prose. If certain views become unavailable in certain modes of operation, or if switching between views is cumbersome, the user's ability to gain insight through multiple perspectives will be constrained.

Allocation to Benefit Hierarchy

Increases individual effectiveness

Expedites routine performance

Accelerates error-free portion of routine performance

If users cannot easily and consistently switch from one view to another in an application and switching is frequently needed to do the task, routine performance will be compromised.

Increases individual effectiveness

Improves non-routine performance

Supports problem solving

If switching between views is easy and fast, users are more likely to look at their work from different viewpoints and may gain additional insight while solving problems. Thus, systems that make views accessible enhance problem-solving.

4.26 Supporting Visualization

A user wishes to see data from a different viewpoint. Systems should provide a reasonable set of task-related views to enhance users' ability to gain additional insight while solving problems. For example, Microsoft Word provides several views to help users compose documents, including Outline and Page Layout modes.

Allocation to Benefit Hierarchy

Increases individual effectiveness

Improves non-routine performance

Supports problem solving

Users can employ multiple views to gain additional insight while solving problems. These multiple perspectives can effectively increase both the quality and quantity of the feedback the user receives. Thus, systems that support visualization enhance problem solving.

5 Software Engineering Hierarchy

This chapter gives the software engineering hierarchy and describes the software engineering mechanisms used in the list of usability scenarios. The hierarchy, in brief, is given in Table 3 and each mechanism listed is described in subsequent sections.

Table 3 Software Engineering Hierarchy

Separation

- Encapsulation of function
- Data from commands
- Data from the view of that data
- Authoring from execution

Replication

- Data
- Commands

Indirection

- Data
- Function

Recording

Preemptive scheduling

Models

- Task
- User
- System

An item that affects the range of these mechanisms is how broadly they are shared. That is, a mechanism that is embedded in the infrastructure and available to any application is more far reaching than a mechanism that is kept within a single application or within a set of applications. We do not capture this range consideration within the description of the mechanisms.

5.1 Separation

Separation is a standard engineering technique for dealing with complicated problems. It involves breaking a complicated problem into two or more distinct portions, solving each portion and then unifying the solution. The success of this technique depends on the difficulty of the unification. If the concerns addressed by each element of the decomposed problem are truly different, then the unification is simple. If not, a large amount of communication between the distinct portions is required to solve the total problem, and the separation technique may have been incorrect. The term *coupling* is used to describe the amount of communication and cooperation necessary to unify the two separated portions. Low coupling indicates that the separation is achieving the goal of making the total problem easier to solve. High coupling indicates that the separation is not achieving that goal.

The essence of separation, then, is to choose portions of the problem that deal with different concerns. The following specific mechanisms describe different portions that may be separated. We begin by describing encapsulation of function. All of the forms of separation we identify actually encapsulate function, but we reserve the term to describe other forms of separation than separation of data from command, data from the view of data, and authoring from execution.

5.1.1 Encapsulation of function

Encapsulation is probably the most basic software engineering mechanism. It means enclosing functionality within a module, exposing only what is necessary to achieve that functionality, and returning results. Everything else is hidden within the module. This is separation because the encapsulated functionality is separated from other functionality. Encapsulation enables a developer to modify the algorithms within the module without changing other portions of the system.

5.1.2 Data from commands

Separating data from function is a mechanism that allows a number of distinct commands to be performed on a set of data, or a single command to be performed on a number of distinct data sets. When using this mechanism, the data (or sets of data) are encapsulated separately from the command or commands. The commands are user-specified commands (or maybe abbreviations or aggregations of user commands). This mechanism is most appropriate when either the set of commands or the set of data are dynamic. That is, the data being operated on by the commands may be highly changeable and the set of commands that the user can specify may be highly changeable.

5.1.3 Data from the view of that data

Separating data from the view of that data is a mechanism that allows distinct perspectives to be placed on a set of data. The data is itself encapsulated into an area with various access and

modification functions, as above. The description of how a user might wish to see that data is also maintained as a distinct collection. It specifies items such as units, language, filters for data items, methods for combined data items, style sheets, and so forth. Separating the data from a description of the view of that data allows different users to express different preferences, allows data to be hidden from certain users, and allows users to view the data differently depending on the platform they are currently using.

5.1.4 Authoring from execution

Separation of the authoring of a specification of an action from the execution of that specification is a basic element of all software development. This separation is an example of the separation of concerns in that the support necessary for authoring a specification is distinct from the support necessary to interact with the result of an execution of that specification. We are interested in a much more restrictive meaning of this separation. We are interested in the aspect of authoring that allows an end user to specify the behavior of a software system within that system. This may be as simple as choosing settings on a menu or as complicated as using a scripting language. The specification may also persist across executions of the system or it may exist for only the current execution. The specification may also be a schedule of particular activities to be executed after terminating the current execution.

Authoring incurs a cost in human behavior. That is, it takes time and effort. Any analysis of the costs and benefits of allowing the end user to author behavior should consider the cost of authoring as well as the benefits that result.

5.2 Replication

Replication is the mechanism of having duplicate copies or variants within the software system of some entity. This entity can be data or it can be function. The general reasons why one would replicate either data or function are to increase performance, to increase reliability, or to provide alternative routes for the achievement of a particular result.

5.2.1 Data

Replication maintains data in several different locations within the system. This reason for the replication is sometimes to increase performance and sometimes to increase availability. One instance of this mechanism for the purpose of improving performance is to cache data in the same structure in several different locations with different access times. For example, a web page may be cached on a local machine to decrease retrieval time from the Internet. Another form of this mechanism is to maintain the data in different structures. For example, large data sets are often maintained with index files that speed up the searching process. One form of this mechanism to improve availability is the saving of state for the purpose of restarting a system in the event of failure.

Regardless of the reason for replication, whenever the same data is found in two locations it is necessary to maintain consistency. That is, regardless of where it is accessed, the data should be the same. There are a variety of schemes that maintain consistency; the important point is to ensure consistency whenever replication is used.

5.2.2 Commands

Commands are replicated in order to provide for multiple user interfaces to achieve the same functionality. These user interfaces could be remote versus local, or it could be that alternative paths are available for an end user to achieve a desired functionality. In any case, different commands may be available to achieve the same goal.

5.3 Indirection

Indirection is the interposition of an intermediary between either data or control accesses. In either case, indirection is another mechanism intended to reduce the coupling between distinct elements.

5.3.1 Data

We will use the terms “data producer” and “data consumer” to describe the data indirection mechanism. A direct connection would have the data producer providing the data directly to the data consumer(s). This means that there is a tight coupling between the data producer and the data consumer(s) and that either knowledge of the consumer is embedded in the producer or vice versa. Either type of knowledge means that the addition or deletion of a data consumer will affect the data producer (or vice versa). By interposing a registration mechanism the coupling can be reduced.

The registration mechanism works by providing a separate component to distribute the data. A consumer would register with the distribution manager that it is interested in a particular data item and a producer would register with the distribution manager that it produces a particular data item. The registration process can be done either at specification time or at execution time. Both the consumer and producer of data have a direct relationship with the distribution manager but not with each other. A new consumer can be added or removed by informing the distribution manager, while the producer remains unaffected.

5.3.2 Function

Indirection consists of putting an intermediary function between various alternative methods of accomplishing a particular service. Terms such as a “virtual device,” a “virtual tool kit,” a “strategy pattern,” and a “factory pattern” describe this mechanism. Binding between the service requester and the alternative service provider service may be done before or at execution time. In any case, the service requester uses a single interface to interact with the func-

tion indirection manager, and the function indirection manager translates the information received specifically for the alternative chosen.

5.4 Recording

This mechanism records system state periodically for further use. Some of the variables that are dependent upon the particular application of the mechanism are

- the frequency with which the state is recorded
- the actual state recorded
- the use to which the recorded state is put
- the persistence of the data recorded. Some applications require that the state is recorded in persistent storage; others require that it be recorded in volatile storage.
- the consistency of the data recorded. In some cases, the data will be consistent because the application interrupts its other activities in order to record data. In other cases, consistency of the data may not matter. In still other cases, a transaction type mechanism may be required in order to guarantee data consistency.

5.5 Preemptive Scheduling

Scheduling is the mechanism whereby resources are assigned to activities within the computer system. The types of resources may be physical, such as memory, central processing unit, input/output peripherals; or they may be logical, such as queues, flags, or other entities.

In general, scheduling can be done on a preemptive or a non-preemptive basis. That is, once an activity has a resource, it may have the resource taken away (preemptive) or it may keep that resource until it voluntarily yields it (non-preemptive). Within these two broad categories are a variety of scheduling mechanisms. The choice of a particular mechanism for a particular resource is based on several considerations including the type of resource, maximizing utilization of the resource, minimizing waiting time for the resource, and priority of one task over another. Measures of a scheduling mechanism include utilization of the resource, worst-case waiting time, average waiting time, and so forth.

Preemptive scheduling allows the software system to have multiple simultaneous activities. In fact, the activities are not simultaneous when examined at a tiny time scale (measured in terms of microseconds) but they appear simultaneous when examined at a larger time scale (measured in terms of 10s of milliseconds). The term *thread* refers to a logical sequence of activities within the computer system. At any point in time, a thread is either active (consuming the processor resource) or blocked (waiting for a resource or for some input). Having multiple simultaneous activities is expressed as having multiple threads. Having multiple threads is most often accomplished (although not exclusively) by using a preemptive processor scheduling strategy.

5.6 Models

The mechanism of keeping a model within the system allows the model to be used for prediction. Different models can predict different types of items. Each model requires various types of input to accomplish its prediction. Clearly identifying the models that the system uses to predict either its own behavior or the user's intention enables designers to tailor and modify those models either dynamically or off-line during development.

5.6.1 Task

In this case, the model maintained is that of the task. The model of the task is used to determine context so the system can have some idea of what the user is attempting to accomplish and can provide various kinds of assistance.

5.6.2 User

In this case, the model maintained is of the user. The model determines the user's knowledge of the system, the user's behavior in terms of expected response time, and other aspects that are specific to a user or a class of users.

5.6.3 System

In this case, the model maintained is that of the system. The model determines the expected behavior of the system so that appropriate feedback can be given to the user. The model of the system predicts items such as the time needed to complete current activity.

6 Architectural Patterns and Categorization

This chapter presents an architectural pattern for each of the general usability scenarios and places each scenario into the software engineering hierarchy. These patterns do not represent the only possibility for implementing the scenarios. Rather, they present one pattern and discuss many of the issues associated with its implementation. Thus, an architect may find these patterns useful even if they are not adopted.

6.1 Aggregating Data

A user may want to perform one or more actions on more than one object. For example, an Adobe® Illustrator® user may want to enlarge many lines in a drawing. It could become tedious to perform these actions one at a time. Furthermore, the specific aggregations of actions or data that a user wishes to perform cannot be predicted; they result from the requirements of each task. Systems, therefore, should allow users to select and act upon arbitrary combinations of data.

6.1.1 Pattern

The module view of an architecture pattern for this scenario is shown in Figure 2.

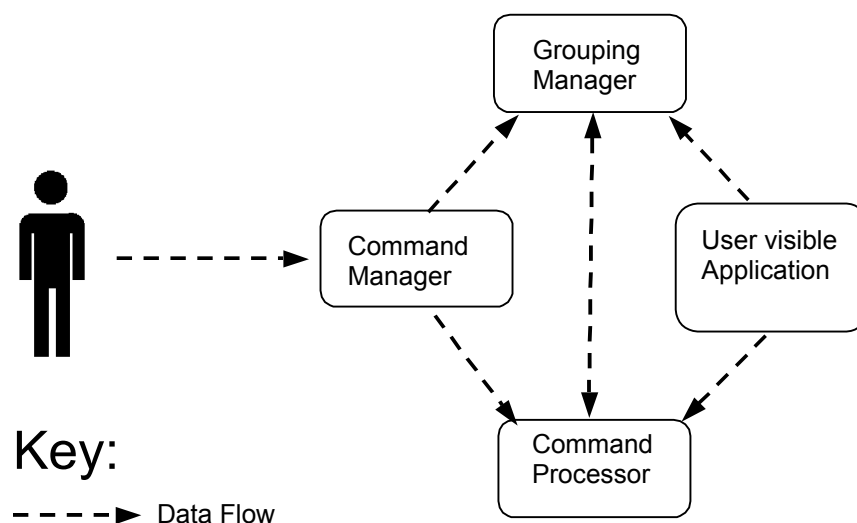


Figure 2. Aggregation of Data Architecture Pattern

This pattern has the following components.

The command manager. This component manages the commands that the user generates. A command has an action and one or more subjects that either provide input or accept output from the command. Feedback about the command is generated by the command manager and passed to the presentation. (The output generated by the command is outside of the scope of this scenario.) There are three types of commands. Some commands are independent of groups and group management. These commands are passed directly to the command processor and executed. These commands are outside of the scope of this scenario. A second type of command is concerned with the management of groups – creating a group, adding data to a group, removing data from a group, or deleting a group. These commands are passed to the grouping manager. A third type of command is an action in which one or more of the subjects is a group. This type of command is either passed to the grouping manager or the command processor. The situations where it is passed to one or the other are discussed below.

The grouping manager. This component manages the definition of groups and the addition and deletion of data items from a group. It should, at a minimum, support commands that create and delete groups, and add to and delete from groups. Both data points and other groups should be able to be added and deleted from groups. In the case described below, the grouping manager controls the iteration of commands through the command processor. It accesses the user-visible data and presents groups to support group editing commands.

User-visible application data. This component provides access to the application data that is visible to the user. The data may reside in a single repository (as we have displayed in Figure 2) or it may be distributed through a collection of components. In any case, the application data visible to the user is available both to those components that control data presentation and those that manipulate the data.

There are two options for applying a command to a group, iteration and embedded grouping. Iteration describes the case in which the particular command operates on single arguments. In this case, the grouping manager manages the application of the command to a group. The grouping manager repeatedly invokes the correct command processor on each item in the group. This option assumes that applying a command to a group means applying the command to each element of the group.

The second option is embedded grouping. In this case, the command processor understands groups and can directly operate on a group. The command and its arguments (including the grouped arguments) are sent directly to the command processor. The grouping manager must make available the group elements to the command processor. This can be done synchronously by having the command processor query the grouping manager upon receiving the

command, or it can be done asynchronously by having the group identification and the group members recorded in a location available to the command processors.

6.1.2 Allocation to mechanism hierarchy

Separation

Data from Commands

In order to allow both the grouping manager and the command processor to manipulate the same data, this data must be kept separate from both components. Such separation also allows presentation commands to read the same data that the grouping manager and command processor manipulate.

Separation

Authoring from Execution

Users author commands for use on aggregates by issuing them to the command manager, but the grouping manager controls iteration in such cases by generating a new set of commands (based on the user's input) and issuing them in sequence to the command processor. Thus, the user's command authoring is kept separate from its execution.

6.2 Aggregating Commands

A user may want to complete a long-running, multi-step procedure consisting of several commands. For example, a psychology researcher may wish to execute a batch of commands on a data file during analysis. It could become tedious to invoke these commands one at a time, or to provide parameters for each command as it executes. If the computer is unable to accept the required inputs for this procedure up front, the user will be forced to wait for each input to be requested. Systems should provide a batch or macro capability to allow users to aggregate commands.

6.2.1 Pattern

The logical view of an architecture pattern for the authoring portion of this scenario is shown in Figure 3.

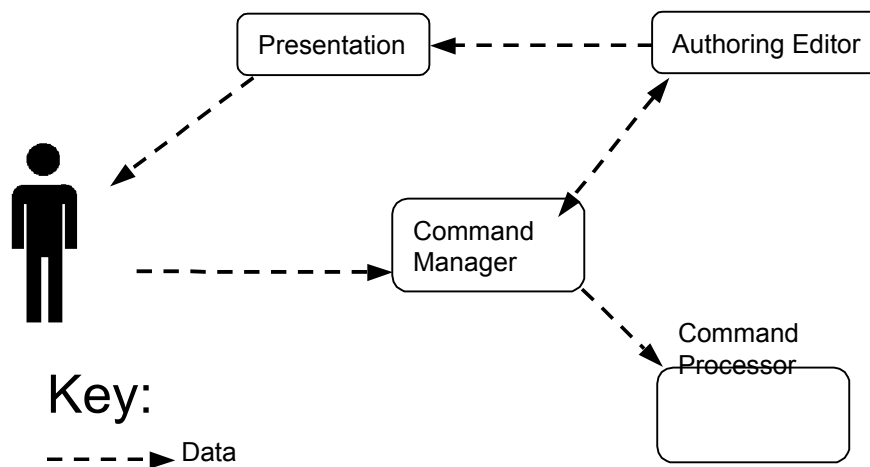


Figure 3. *Authoring of Aggregation of Commands Architecture Pattern*

There are two possibilities for authoring aggregates of commands. The first possibility is to have a separate editor that supports this authoring. In this case, the relevant components are the command manager and the authoring component. The command manager recognizes invocation of the authoring editor and those commands that are intended for the authoring editor. The editor creates and stores the aggregated commands. Feedback to the user is through the presentation component. The following items must be specified: Where should output generated during the execution of the aggregated commands be written, and where does input come from that normally would be provided by the user synchronously during execution?

The second architecture possibility is to author by demonstration. That is, the user executes the commands that are desired synchronously. The authoring editor monitors those commands and saves them as an aggregated set that can be subsequently invoked. These aggregated commands are usually edited prior to final saving for execution. In this case, the command manager must communicate the commands both to the command processor for execution and to the authoring editor for inclusion in the aggregate. The authoring editor and the command processor may communicate if additional information about parameters or data must be saved in the aggregated command. We do not display this communication since it may not be necessary.

The execution of the aggregates can also be explained using Figure 3. In this case, the user invokes the aggregated command. The command manager must communicate with the authoring editor to retrieve the aggregated command. (This could be as simple as retrieving it from a file where the authoring editor has stored it). This necessity for communication is the source of the data flow from the authoring editor to the command manager. The command manager then sends the commands one at a time to the command processor. The command manager must be informed of the source of any required synchronous user input and the destination of any generated error messages.

6.2.2 Allocation to mechanism hierarchy

Separation

Authoring from Execution

Users specify aggregates by authoring them in the authoring editor. Since the command manager controls the execution of the commands, authoring and execution are separated.

Replication

Commands

More than one set of aggregated commands might be applied by a user to achieve the same goals. Commands are effectively replicated in this case.

Recording

In systems where demonstration is used to specify command aggregates (e.g., macros in Microsoft Word), the authoring editor must be able to record the user's actions.

6.3 Canceling Command

A user invokes an operation, then no longer wants the operation to be performed. The user now wants to stop the operation rather than wait for it to complete. It does not matter why the user launched the operation. The mouse could have slipped. The user could have mistaken one command for another. The user could have decided to invoke another operation. For these reasons (and many more), systems should allow users to cancel operations.

6.3.1 Pattern

We first present the module view that enumerates the components involved and their responsibilities. We then present a sequence chart that shows how the components interact.

Figure 4 presents the module view of the cancellation architectural style. It has the following components together with their responsibilities. (Note that these functions may or may not be implemented as separate software components, such as classes or threads.)

1. Activity Component. Activity components perform the activities that may be cancelled. They must cooperate with the controller to provide information about the resource that they use and their collaborations. They must also have a mechanism for retaining sufficient information about the system state be able to restore that state at any time. The mechanism will be exercised by the cancellation component but the active components must prepare resources so that the cancellation component can, in fact, exercise the mechanisms.

2. Cancellation Listener. This component listens for the user to request canceling the active components. It must inform the user that it has received the cancellation request. It then in-

forms the cancellation controller. If necessary, it may spawn a new thread to control the operation of the cancellation component.

3. *Cancellation Controller*. This component can terminate the active thread, return the persistent resources to their state prior to invoking the active components, release non-preemptable resources, provide feedback to the user about progress and the result of the cancellation, and inform collaborating components of the termination of the active thread. It is responsible for gathering information about the resources used by the active components and the collaborating components.

4. *Collaborators*. These components are responsible for receiving information about the termination of the active components.

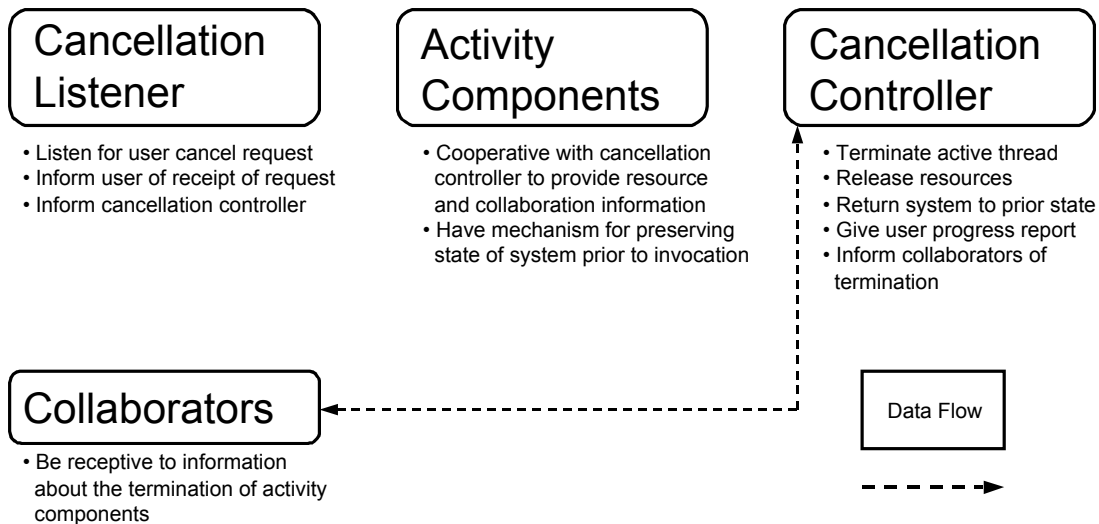


Figure 4. Module View of Cancellation Architecture Pattern

Figure 5 shows a representation of the threads of this pattern. The following logically distinct threads of activity exist, although they may be implemented together.

- *The listener thread*. This thread interacts with the user. It provides the user with a means to indicate what is to be cancelled.
- *The cancellation control thread*. This thread manages the cancellation activities. It must be independent from the active thread.
- *The active thread*. This is the running thread that the user wishes to cancel. The activities running under the control of the active thread are those that are to be cancelled.
- *The collaborating processes thread*. This may be a collection of threads but we model it as a single thread. This thread manages those processes that collaborate with the active

thread.

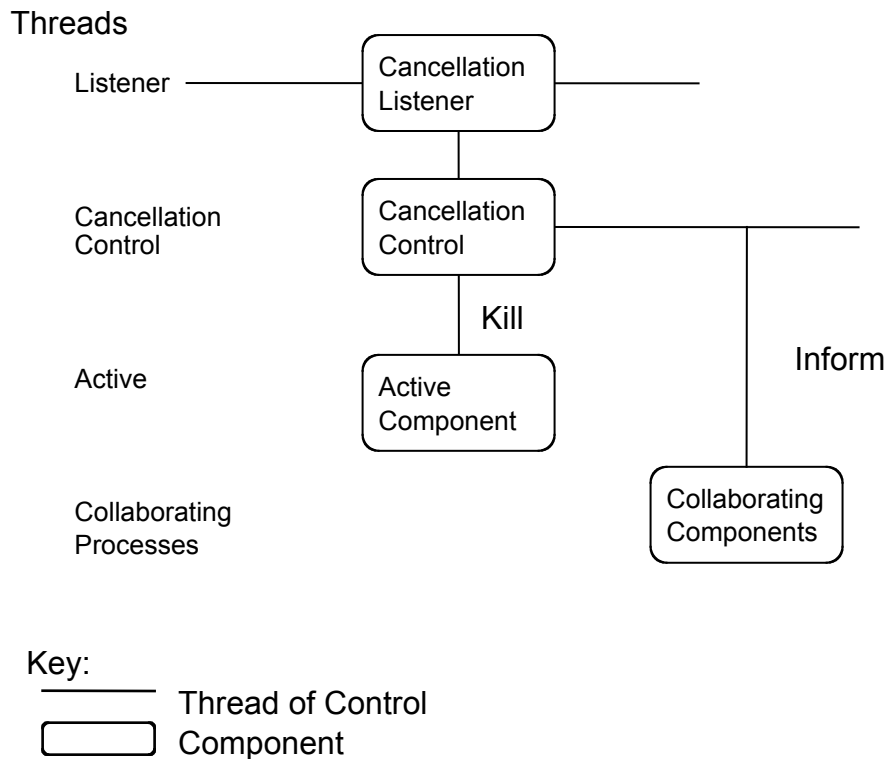


Figure 5. Cancellation Pattern - Thread View

We will now describe how these threads interact. The user sends a cancellation stimulus to the cancellation listener component running in the listener thread. This thread then provides feedback to the user that the cancel request has been received, and lets the cancellation controller know that it should carry out that the cancel activity.

The cancellation controller is a thread that executes the cancellation. The active thread should not be expected to listen for a cancel request since it may be blocked for some reason or it may be in an infinite loop. The cancellation component carries out four activities:

1. Terminate the active thread.
2. Inform the user of the progress and results of the cancellation.
3. Return the system to its state prior to the invocation of the active thread. This involves
 - a. restoring any persistent resources to their state prior to the invocation of the active thread
 - b. releasing non-preemptive resources acquired by the active thread
4. Inform threads collaborating with the active thread that it has been terminated.

The first of these is straightforward. Presumably, the thread control mechanisms of the operating system permit terminating a thread. The second activity is also straightforward. The user should be informed of both the progress and the results of the cancellation. The third activity (returning to prior state) requires that the cancellation controller be aware of the mechanisms for restoring persistent resources to their states prior to invoking the active thread. For example, operating system resources may need to be reallocated, files may need to be restored, network connections may need to be re-established, etc. The cancellation controller must also be aware of the resources acquired by the active components, and these resources must be freed. This awareness can be achieved by a variety of mechanisms. The activity components can report resources acquired to the cancellation controller, the cancellation controller can intercept requests for resources, or the various resource managers can provide this information to the cancellation controller.

The fourth activity (informing collaborating threads) also requires knowledge on the part of the cancellation controller. The cancellation controller must be informed of collaborations, either synchronization or data communication, that the active thread has with other threads. The controller doesn't necessarily need to be informed of the state of these collaborations; it can simply inform each of the collaborating threads of the termination of the active thread. Then it becomes the responsibility of the collaborating threads to perform the correct actions, including providing information to the user about the progress and results of these cancellation requests. This can be complicated. It depends on the type of collaboration and the extent to which the collaborating components depend on the completion of the active components. One possibility is to treat the information as a cancellation request for the collaborating components. In this case, a recursive use of the pattern will achieve the desired results. Other types of collaborations may not require cancellation. In any case, a decision must be made as to the desired result after the collaboration components have been informed of the cancellation of the active components.

6.3.2 Allocation to mechanism hierarchy

Preemptive Scheduling

To adequately implement cancellation, the cancellation listener and cancellation controller must occupy independent threads.

Models System

After a command has been cancelled, the system must consult an explicit model of itself in order to predict state restoration time and to report progress.

Recording

The cancellation component must record its initial state so that the system can be returned to the state prior to the invocation of the cancelled components.

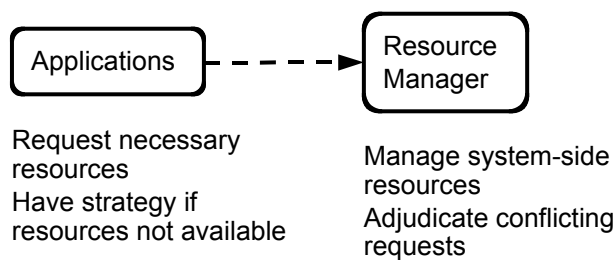
6.4 Using Applications Concurrently

A user may want to work with arbitrary combinations of applications concurrently. These applications may interfere with each other. For example, some versions of IBM® ViaVoice and Microsoft® Word contend for control of the cursor with unpredictable results. Systems should ensure that users can employ multiple applications concurrently without conflict.

(See: Supporting Multiple Activities).

6.4.1 Pattern

Figure 6 gives a module view of a possible pattern.



Key:

- - - -> Data Flow

□ Component

Figure 6. Module View of Concurrent Application Use

This pattern describes a system resource manager that applications must use to get shared resources such as the cursor or memory. The applications are responsible for requesting necessary resources from this resource manager and then querying for state of shared resources when they are being used. The resource manager adjudicates conflicting requests for resources and manages all system resources, whether sharable or only serially usable.

6.4.2 Allocation to mechanism hierarchy

Separation

Encapsulation of function

The resource manager is kept separate from other applications.

Models

System

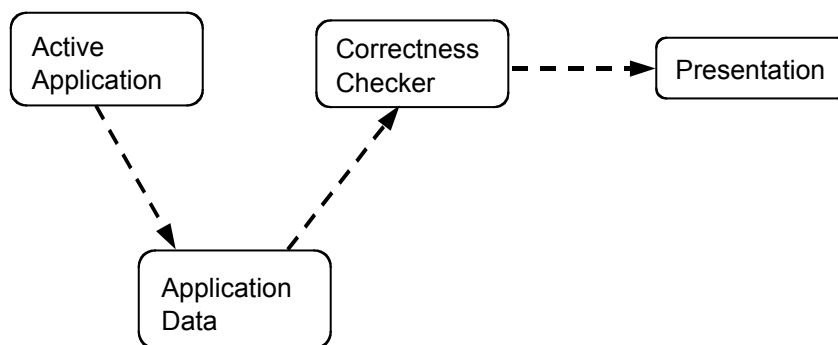
The resource manager maintains an explicit model of the system to track resources and their consumers.

6.5 Checking for Correctness

A user may make an error that he or she does not notice. However, human error is frequently circumscribed by the structure of the system; the nature of the task at hand, and by predictable perceptual, cognitive, and motor limitations. For example, users often type “hte” instead of “the” in word processors. The frequency of the word “the” in English and the fact that “hte” is not an English word, combined with the frequency of typing errors that involve switching letters typed by alternate hands, make automatically correcting to “the” almost always appropriate. Computer-aided correction becomes both possible and appropriate under such circumstances. Depending on context, error correction can be enforced directly (e.g., automatic text replacement, fields that only accept numbers) or suggested through system prompts.

6.5.1 Pattern

Figure 7 gives a module view of an architecture pattern for a correctness checker. The application data is maintained separately from the application so that it is accessible to the correctness checker. The correctness checker maintains a model of the correct input so that it can determine when a potential error occurs. In the sample pattern, it reports this error to the user through the presentation. There may be cases when errors are automatically corrected.



Key:
- - - -> Data Flow

Figure 7. Correctness

This figure assumes that the active application and the correctness checker operate on different threads of control. In this way, error detection can be done without user input. It also can be done asynchronously, although this is not displayed in the module view.

6.5.2 Allocation to mechanism hierarchy

Separation

Data from Commands

When the correctness checker and the data operate independently, a variety of different checking mechanisms can be turned on or off without affecting the data itself.

Preemptive Scheduling

Correctness must be checked while the user performs operations. This requires an independent thread.

Models

Task

Systems can employ a model of the task to identify when, what, and how to correct. For example, knowing that sentences usually start with capital letters would allow an application to correct a lower case letter that begins a sentence.

Models

User

Systems can employ a model of the user to identify when, what, and how to correct. For example, a model of human typing is used to know that “teh” is a common mistyping of “the.”

Models

System

A system can employ a model of itself to identify when, what, and how to correct. For example, knowing which toolbar buttons are adjacent might help the system conclude that a user hit the wrong button when a tool selection doesn’t make sense according to context.

6.6 Maintaining Device Independence

A user attempts to install a new device. The device may conflict with other devices already present in the system. Alternatively, the device may not function in certain specific applications. For example, a microphone that uses the Universal Serial Bus (USB) may fail to function with older sound software. Systems should be designed to reduce the severity and frequency of device conflicts. When device conflicts occur, the system should provide the information necessary to either solve the problem or seek assistance. (Devices include printers, storage/media, and I/O apparatus.)

6.6.1 Pattern

Figure 8 shows a module view of a virtual device layer. Applications access physical devices through a virtual device that defines and abstracts how the devices should be controlled. The virtual device layer accesses devices through physical device drivers that do the actual con-

trol. The virtual device layer translates the virtual device into the various physical devices. This assumes that a collection of device types has been defined and that virtual device layers exist for each category of device types.

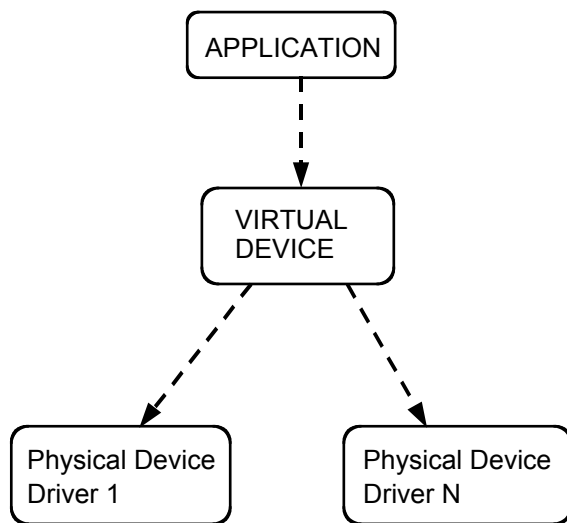


Figure 8. Virtual Device Layer

6.6.2 Allocation to mechanism hierarchy

Indirection Function

Commands for interacting with various devices can be abstracted. Such indirection hides distinctions between physical devices by providing an encapsulated virtual device interface.

Models System

The system can maintain a model of itself based on data from the virtual device interface to identify and track device conflicts and to provide the information needed to resolve them.

6.7 Evaluating the System

A system designer or administrator may be unable to test a system for robustness, correctness, or usability in a systematic fashion. For example, the usability expert on a development team might want to log test users' keystrokes, but may not have the facilities to do so. Systems should include test points and data gathering capabilities to facilitate evaluation.

6.7.1 Pattern

Figure 9 shows the module view of a data-recording mechanism. This module provides a means for recording any data of interest. Recorded data may include keystrokes, errors, user commands, or any other data of interest.

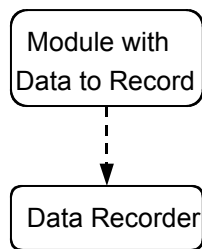


Figure 9. Data Recording

There should be tools that can access the recorded data and provide a variety of different analyses, such as examining the data for patterns of errors, user confusion, or critical incidents. The analysis tools associated with the recording of data are not shown.

6.7.2 Allocation to mechanism hierarchy

Indirection Data

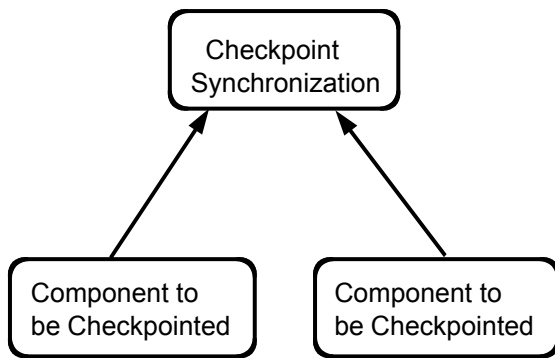
When evaluating a system, test data may need to “masquerade” as user data for simulation purposes. Data indirection can be used to facilitate this process.

Recording

The data recorder records the data needed to analyze the results of test cases and other evaluations.

6.8 Recovering from Failure

A system may suddenly stop functioning while a user is working. Such failures might include a loss of network connectivity or hard drive failure in a user’s PC. In these or other cases, valuable data or effort may be lost. Users should be provided with the means to reduce the amount of work lost from system failures.



Key:

→ Control Flow

Figure 10. Perform Checkpoint

6.8.1 Pattern

There are elaborate mechanisms to automatically recover from failures that involve redundant computations. These mechanisms are intended to reduce down time for systems that require high availability. We here describe a mechanism that is intended to apply in situations in which the availability requirement is less stringent.

The mechanism maintains periodic checkpoints of application state. That is, the data necessary for the application to execute is recorded when the system is in a consistent state. The system then has a restart mode in which it reads the recorded data and resumes computation from the last state recorded. Figure 10 above shows the key components. Each component is responsible for checking its data and restoring its state from the checked data. This can be done individually by the components or globally through a data repository or some mixture. The important aspect is that the checked data be consistent. This is the role of the checkpoint synchronizer. It informs the components that maintain application data when it is time for a checkpoint and ensures that this action is performed with consistent data.

6.8.2 Allocation to mechanism hierarchy

Recording

The checkpoint synchronizer informs components to record all relevant data to minimize loss in the event of failure.

6.9 Retrieving Forgotten Passwords

A user may forget a password. Retrieving and/or changing it may be difficult or may cause lapses in security. Systems should provide alternative, secure mechanisms to grant users access. For example, some online stores ask each user for a maiden name, birthday, or the name of a favorite pet in lieu of a forgotten password.

6.9.1 Pattern

The key element of the pattern is to have authentication encapsulated. There are a variety of policies and mechanisms that can be used for authentication. These include secret questions, mailing a new password to a previously authenticated mail address, and biologically-based mechanisms such as finger print recognition or retinal scan. The policy and mechanism to be used depends on the level of security to be implemented and the context of use. All data relating to authentication should be stored using encryption so the information is not available if the storage media is compromised.

6.9.2 Allocation to mechanism hierarchy

Separation

Encapsulation of function

The authentication component must be encapsulated and passwords encrypted to prevent security breaches if the media that stores it becomes compromised.

6.10 Providing Good Help

A user needs help. The user may find, however, that a system's help procedures do not adapt adequately to the context. For example, a user's computer may crash. After rebooting, the help system automatically opens to a general table of contents rather than to a section on restoring lost data or searching for conflicts. Help content may also lack the depth of information required to address the user's problem. For example, an operating system's help area may contain an entry on customizing the desktop with an image, but may fail to provide a list of the types of image files that can be used. Help procedures should be context dependent and sufficiently complete to assist users in solving problems.

6.10.1 Pattern

In the pattern, we focus on achieving context dependency. The content and verbosity of the help messages is not an architectural issue. Having context dependent help requires two types of knowledge: knowledge of what the user wishes to do and knowledge of the current state of the system. Knowledge of the current state of the system can be gathered explicitly. Knowledge of what the user wishes to do, however, must be inferred. This inference comes from what the user has already done, the current system state, and a model of the tasks the user

might wish to do. The task model should be separated from other functions of the system, since it is likely to change as it is refined. In other words, there should be an explicit task model developed that supports context dependent help.

Figure 11 presents the pattern. The context determiner is responsible for determining the current context so that the help system can show the correct information. The system state provides input to the context determiner without specifying its source. Finally, the task model is a separate encapsulated component, for the reasons we have already mentioned. We do not show the run-time interactions, but there are two cases. If the context dependent help is automatically invoked, it should run in a thread that is separate from the thread of the user activity. If context dependent help depends on user invocation, then its thread structure can be synchronous with the user's request.

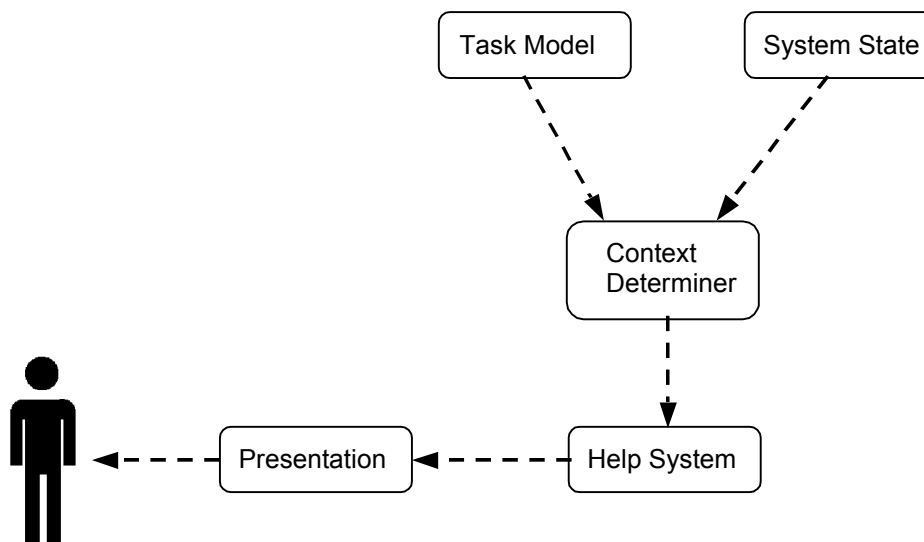


Figure 11. Context Dependent Help

6.10.2 Allocation to mechanism hierarchy

Preemptive Scheduling

If a system is to offer help automatically when it perceives that a user is in trouble, the system must support scheduling; the help system and context determiner will occupy an independent thread.

Models

Task

A model of the task can be used to determine what help information to present first when either the user requests help or the help system determines that the user is in trouble.

Models

User

A model of the user can be used to determine what level of help to provide and how likely the user is to encounter problems (i.e., is the user a novice or an expert?). This model can also help determine what information to present.

6.11 Reusing Information

A user may wish to move data from one part of a system to another. For example, a telemarketer may wish to move a large list of phone numbers from a word processor to a database. Re-entering this data by hand could be tedious and/or excessively time-consuming. Users should be provided with automatic (e.g., data propagation) or manual (e.g., cut and paste) data transports between different parts of a system. When such transports are available and easy to use, the user's ability to gain insight through multiple perspectives and/or analysis techniques will be enhanced.

6.11.1 Pattern

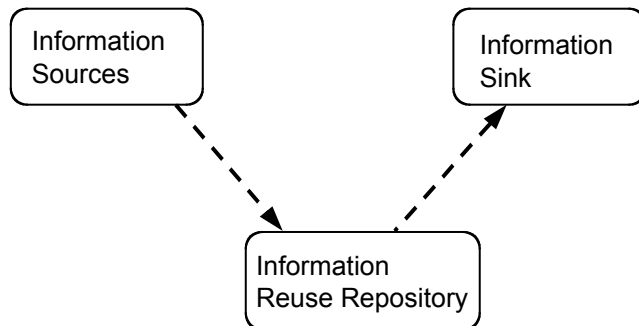
Figure 12 shows the components that implement information reuse. The interpretation and persistence of these components, however, depend on the type of information reuse being implemented. We discuss two cases, automatic and manual.

If manual cut and paste is being implemented, then the user (somehow) indicates the data to be transferred. This data is, typically, available on the display, so the information source is the presentation component. The information is then placed in the information reuse repository. Finally, the user indicates the sink for the data and the data is passed to the source as if it were input data. One key to cut and paste is to maintain type information about the data. The source provides the type of the data being transferred and the sink indicates whether it can accept data of that type. If not, then a type conversion must be made between the type of the data at the source and an acceptable type at the sink. Usually, system-wide conventions concerning type have been established so the source and sink can accommodate the same types.

If automatic data propagation is being implemented, then type information is again a crucial element. In this case, the information reuse repository is maintained as a blackboard. The sink(s) register interest in items of a particular data type. The source publishes current information of the particular data type. For example, if the sink is a form that wants to know the user name, any component of the system that knows it places that name in the repository. When the sink is invoked, it registers for the user name and the current name is given to the sink.

The type of information used in data propagation is far richer than the type used in cut and paste. A name, for example, in cut and paste can be treated as a sequence of letters. In data propagation, the name should be identified as "proper name" or some other type. Usually, a

data dictionary is defined that enumerates all the valid types known to the information reuse repository.



Key:

- - - -> Data Flow

Figure 12. Information Re-Use

6.11.2 Allocation to mechanism hierarchy

Indirection Data

The information reuse repository acts as an indirection intermediary by separating the data producer and consumer.

6.12 Supporting International Use

A user may want to configure an application to communicate in his or her language or according to the norms of his or her culture. For example, a Japanese user may wish to configure the operating system to support a different keyboard layout. However, an application developed in one culture may contain elements that are confusing, offensive, or otherwise inappropriate in another. Systems should be easily configurable for deployment in multiple cultures

6.12.1 Pattern

Figure 13 gives the module view of the pattern. Some of the complications are screen real estate management, screen painting, and deployment strategy.

Screen real estate management

The same phrase in different languages may require a different amount of characters even when the languages utilize the same alphabet. Thus, screen layout may be different in differ-

ent languages. One technique for managing this is to have a template supply the screen layout. This template would be interpreted at runtime by the presentation component. The template is accessible through the customization component.

Screen painting

Different languages may use a different order of painting. Latin-based languages are read from left to right. Hebrew and Arabic languages are read right to left. Some Oriental languages are read top to bottom. It is the responsibility of the presentation component to paint the screen and retrieve input in the correct fashion for the current user.

Deployment strategy

The architect must decide whether to have a single deployment support multiple languages (such as an ATM machine) or whether each deployed system only supports a single language (such as software intended for use within an office where the users' language is known). If a single deployment supports multiple languages, all of the language dictionaries must be present and the presentation must decide on the layout at runtime. This decision is made based on the user's identity or expressed preference. If a single deployment only supports one language then only the dictionary for that language needs to be present, and different versions of the presentation component can be created.

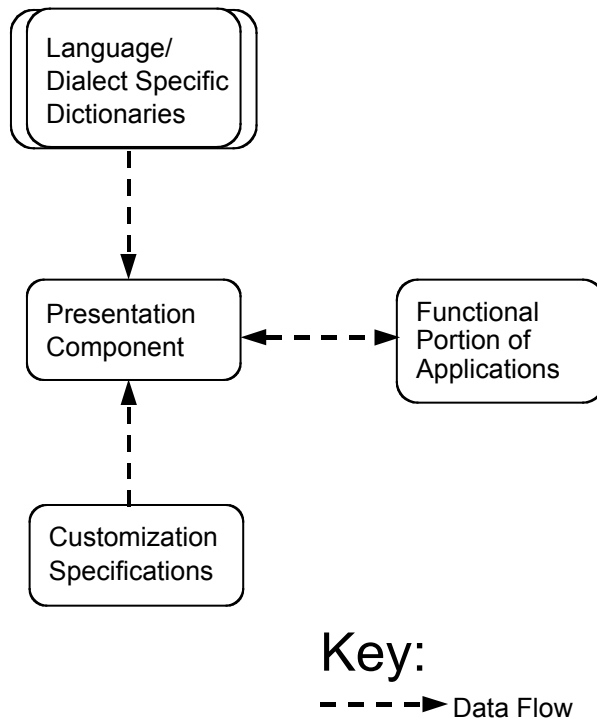


Figure 13. Internationalization

6.12.2 Allocation to mechanism hierarchy

Separation

Data from view of data

By separating the core data of a system from the view of that data, the presentation component can map user-visible data into a form that is linguistically and culturally appropriate.

Models

User

The presentation component accesses user-supplied customization specifications to configure itself appropriately. These specifications model the user.

6.13 Leveraging Human Knowledge

People use what they already know when approaching new situations. Such situations may include using new applications on a familiar platform, a new version of a familiar application, or a new product in an established product line.

New approaches usually bring new functionality or power. When, however, users are unable to apply what they already know, a corresponding cost in productivity and training time is incurred. For example, new versions of applications often assign items to different menus or change their names. As a result, users skilled in the older version are reduced to the level of novices again, searching menus for the function they know exists.

System designers should strive to develop upgrades that leverage users' knowledge of prior systems and allow them to move quickly and efficiently to the new system.

6.13.1 Pattern

There are two facets to this scenario. One facet is platform standards. The user can apply knowledge from one application when using another. The second facet is to have multiple interfaces when making substantive changes to an application. The user can then choose an interface. We discuss these in turn.

Platform standards

The essence of maintaining platform standards is to separate the look and feel of an application or command from the functional aspects of that command. There can then be libraries that implement a standard look and feel for a particular platform. The approach to this separation is discussed in the scenario "Modifying Interfaces."

Multiple interfaces

Figure 13 gives a module view of maintaining multiple interfaces for the purpose of supporting internationalization. Having multiple interfaces for the purpose of backward compatibility can be viewed as simultaneously supporting two languages. One difference is that the two interfaces may not support exactly the same set of commands. This structure will support the case where the commands available through the new interface are a superset of the commands available through the old.

6.13.2 Allocation to mechanism hierarchy

Separation

Encapsulation of function

Toolkits and libraries are encapsulated to embody the standards and the Application Programming Interface (API) of a given platform.

Separation

Data from view

By separating the core data of a system from the view of that data, multiple interfaces can be offered to facilitate the transition from one version of a product to another.

6.14 Modifying Interfaces

Iterative design process is the lifeblood of current software development practice, yet a system developer may find it prohibitively difficult to change the user interface of an application to reflect new functions and/or new presentation desires. System designers should ensure that their user interfaces can be easily modified.

6.14.1 Pattern

Two basic techniques to support modifiability are indirection (both data and control) and encapsulation (information hiding) of coherent computations. Data indirection techniques include repositories and publish/subscribe. Control indirection techniques include virtual machines. This is too complicated a topic to be easily summarized here. See Chapter 6 of *Software Architecture in Practice* for a discussion of several user interface reference models [Bass 1998].

To support modifiability, the architect should enumerate a list of likely change scenarios and ensure that the architecture will support them.

6.14.2 Allocation to mechanism hierarchy

Separation

Encapsulation of function

Encapsulating all user interface functionality away from the core of the system (“Everything Else”) allows designers to modify the user interface more easily.

Indirection Function

The use of an intermediary such as the dialogue manager in the Seeheim Model or the Control in the PAC model is indirection of function. These models are discussed in Chapter 6 of *Software Architecture in Practice* [Bass 1998].

Indirection Data

Indirection of data refers to the separation of application data from the view of that data. It occurs in virtually all of the models discussed in Chapter 6 of *Software Architecture in Practice* [Bass 1998].

6.15 Supporting Multiple Activities

Users often need to work on multiple tasks more or less simultaneously (e.g., check mail and write a paper). A system or its applications should allow the user to switch quickly back and forth between these tasks.

6.15.1 Pattern

The operating system should provide for independent threads of control and for context changes between one thread and another. One problem arises because the operating system must know what an application considers its context. Thus, key bindings as shortcuts may differ from application to application, and the current key bindings should always reflect the active application. Applications should register items so they can be saved and restored when a context switch is made.

6.15.2 Allocation to mechanism hierarchy

Separation Encapsulation of function

The context saver separates application specific environment data like key bindings from the applications themselves.

Preemptive Scheduling

Systems that support multiple threads allow for robust, fast context switching.

6.16 Navigating Within a Single View

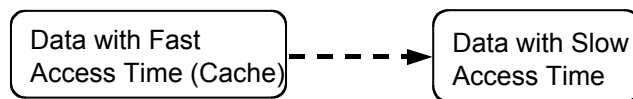
A user may want to navigate from data visible on-screen to data not currently displayed. For example, he or she may wish to jump from the letter “A” to the letter “Q” in an online encyclopedia without consulting the table of contents. If the system takes too long to display the new data or if the user must execute a cumbersome command sequence to arrive at her or his destination, the user’s time will be wasted. System designers should strive to ensure that users can navigate within a view easily and attempt to keep wait times reasonably short.

(See: Working at the User’s Pace)

6.16.1 Pattern

Caching improves performance by replicating data in a location with faster retrieval time. For example, browsers cache copies of Web pages so users can return to those pages quickly.

Figure 14 shows two components that are used to manage caching. One component holds the data with fast access as well as an algorithm that is used to update that data. Usually the cache uses fewer resources than the main data store and so holds a smaller amount of data; This makes it necessary to have an updating algorithm. If the data in the cache is changed by the application, there must also be an algorithm for updating the main data store.



Key:

- - - -> Data Flow

Figure 14. Navigation

6.16.2 Allocation to mechanism hierarchy

Replication Data

By replicating data from a device with slow retrieval times and storing it in a high-speed cache, navigation wait times can be reduced.

6.17 Observing System State

A user may not be presented with the system state data necessary to operate the system (e.g., uninformative error messages, no file size given for folders). Alternatively, the system state

may be presented in a way that violates human tolerances (e.g., is presented too quickly for people to read. See: Working at the User's Pace). The system state may also be presented in an unclear fashion, thereby confusing the user. System designers should account for human needs and capabilities when deciding what aspects of system state to display and how to present them.

A special case of Observing System State occurs when a user is unable to determine the level of security for data entered into a system. Such experiences may make the user hesitate to use the system or avoid it altogether.

6.17.1 Pattern

Two aspects to viewing the system state are making the data available and taking the initiative in presenting the data.

Making the data available

Data that represents the system state should be collected into a component that can make it accessible to the user. Associated with each data item should be information such as type of data, a refresh rate (see Working at the User's Pace), and other attributes of interest. Once the data has been collected, it can be accessed through commands and specialized applications.

Taking the initiative in presenting data

If the data is to be presented upon user request, then no special architecture mechanism is needed. If the data is to be presented on the system's initiative, then a model of the user is needed in order to decide what circumstances will require the system to take initiative.

6.17.2 Allocation to mechanism hierarchy

Separation

Data from Commands

State data is stored in a repository apart from the rest of the system.

Preemptive Scheduling

If data is to be presented on the system's initiative, a component must occupy a separate thread to monitor the user's actions and determine when to present new data or update the data currently displayed.

Models

Task

If data is to be presented on the system's initiative, the system can consult a model of the task to determine what information to present to the user.

Models **User**

If data is to be presented on the system's initiative, the system can consult a model of the user to determine what information to present.

Models **System**

If data is to be presented on the system's initiative, the system can consult a model of itself to determine what information to present.

6.18 Working at the User's Pace

A system might not accommodate a user's pace in performing an operation. This may make the user feel hurried or frustrated. For example, ATMs often beep incessantly when a user "fails" to insert an envelope in time. Also, Microsoft Word's scrolling algorithm does not take system speed into account and becomes unusable on fast systems (the data flies by too quickly for human comfort). Systems should account for human needs and capabilities when pacing the stages in an interaction. Systems should also allow users to adjust this pace as needed.

6.18.1 Pattern

The system maintains a model of itself as well as a model of the user. It uses the model of the user to determine the correct pace or users expectations. It uses to model of itself to predict information that will be of interest to the user, such as time remaining to complete a task.

These models should operate on a thread distinct from the main activity thread so the computations and displays associated with pace control can be completed regardless of the state of the main activity computation.

6.18.2 Allocation to mechanism hierarchy

Models **Task**

The system can use a model of the task to determine what pace to use for prompting. For example, an ATM should know that if a user has asked to deposit money, it might take her or him a while to fill and seal the envelope. The system should not beep at the user during this time.

Models **User**

The system can also use a model of the user to set the pace. For example, novice users may take longer to perform tasks than experts.

Preemptive Scheduling

The model of the user and model of the task should operate on a thread distinct from the main activity thread so the computations and displays associated with pace control can be completed and updated regardless of the state of the main activity.

6.19 Predicting Task Duration

A user may want to work on another task while a system completes a long running operation. For example, an animator may want to leave the office to make copies or to eat while a computer renders frames. If systems do not provide expected task durations, users will be unable to make informed decisions about what to do while the computer “works.” Thus, systems should present expected task durations.

6.19.1 Pattern

See Working at the User’s Pace.

6.19.2 Allocation to mechanism hierarchy

Models Task

A model of the task can determine what steps are likely to be performed in an operation. The system then can predict total completion time that spans more than one operation. For example, if users install several parts of an application in sequence, the system can predict a total install time.

Models System

A model of the system can be used to determine how much work is left to do and, hence, to predict completion time (percent complete, etc.).

Preemptive Scheduling

Scheduling is necessary in systems that present progress feedback. One thread is needed for the progress monitor and one is needed for the activity being monitored.

6.20 Supporting Comprehensive Searching

A user wants to search some files or some aspects of those files for various types of content. For example, a user may wish to search text for a specific string or all movies for a particular

frame. Search capabilities may be inconsistent across different systems and media, thereby limiting the user's opportunity to work. Systems should allow users to search data in a comprehensive and consistent manner by relevant criteria.

6.20.1 Pattern

The architecture pattern for comprehensive search is shown in Figure 15. It uses a search registry to satisfy the search requests. Various applications that have searchable information can register with the search registry. Thus, a file manager might register with the search registry so it can search file names and content. An application requests a search by specifying the types of information that should be reviewed and the scope of the review, the search manager then delegates the search request to the entities that have registered with it. Results are sent back to the application that originally requested the search.

Returning information to the application must allow for appropriate feedback to the user. For example, if the search is for a word in a document, the return should be a reference to that word rather than the word itself so the user can see the word highlighted within context.

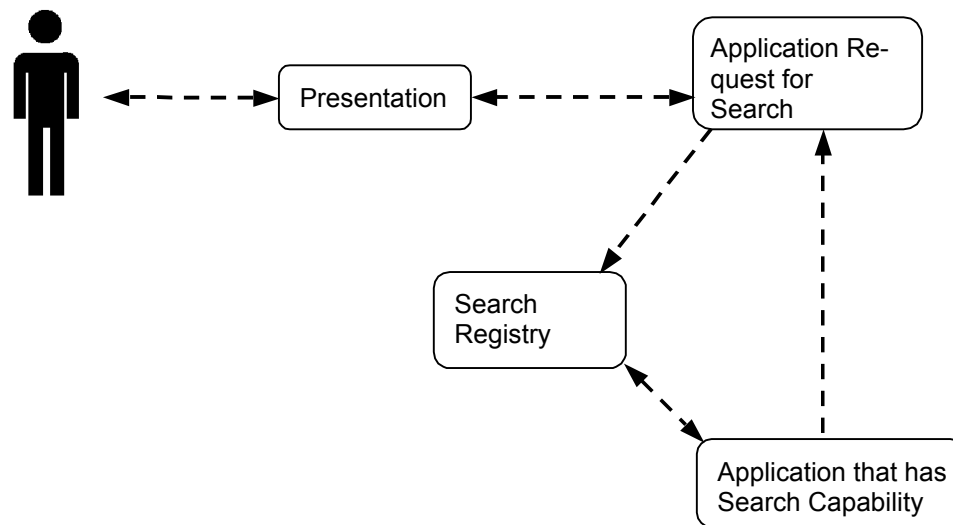


Figure 15. Search

6.20.2 Allocation to mechanism hierarchy

Separation

Encapsulation of function

The search registry is separated from individual applications and files to allow users to instantiate a search from within any application.

Indirection

Function

The search registry operates as an intermediary between the applications requesting searches and the applications executing them.

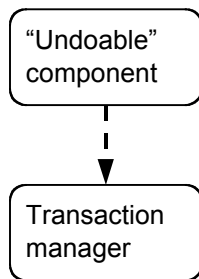
6.21 Supporting Undo

A user performs an operation, then no longer wants the effect of that operation. For example, a user may accidentally delete a paragraph in a document and wish to restore it. The system should allow the user to return to the state before that operation was performed. Furthermore, it is desirable that the user then be able to undo the prior operation (multi-level undo).

6.21.1 Pattern

Figure 16 shows the key components in implementing an undo capability. Each component that is involved in an undo sends relevant data to a transaction manager. The data is viewed as a transaction. That is, data items are grouped together and form an atomic unit. All of the data has been applied or none of it has been applied. (Transactions can be rolled off and that is “undo.”)

Using a global transaction manager rather than relying on each component to perform its own undo has an advantage: the number of steps that can be undone is arbitrary. The only limit is the amount of storage on the media used for the undo capability. Furthermore, the granularity of the undo (e.g., are keystrokes undone or commands) is then dependent on the component rather than on some system-wide decision. That is, suppose the current granularity of the undo is at the command level and there is a decision made to change it to the keystroke level. Then all that is necessary is to enter the keystrokes into the transaction manager and the commands do not need to be modified. The transaction stack will go from just having the data to undo the commands to having the data to undo the keystrokes.



Key:

- - - -> Data flow

Figure 16. Undo

6.21.2 Allocation to mechanism hierarchy

Recording

The transaction manager must store state information each time an “undoable” command is executed in order to later “roll back” through one or more previous states.

6.22 Working in an Unfamiliar Context

A user needs to work on a problem in a different context. Discrepancies between this new context and the one the user is accustomed to may interfere with the ability to work. For example, a clerk in business office A wants to post a payment for a customer of business unit B. Each business unit has a unique user interface, and the clerk has only used unit A’s previously. The clerk may have trouble adapting to business unit B’s interface (same system, unfamiliar context.) Systems should provide a novice (verbose) interface to offer guidance to users operating in unfamiliar contexts.

6.22.1 Pattern

This pattern requires maintaining two different interfaces simultaneously for the same system. This is the same pattern described in Supporting Internationalization and Leveraging Human Knowledge.

6.22.2 Allocation to mechanism hierarchy

Separation

Data from View

The “verbose” and “standard” user interfaces are kept separate from the functional core of the system to easily switch between them.

Replication Commands

Since two user interfaces are offered, commands that are available in both are replicated.

Models User

The system consults a model of the user to determine which interface to present.

6.23 Verifying Resources

An application may fail to verify that necessary resources exist before beginning an operation. This failure may cause errors to occur unexpectedly during execution. For example, some versions of Adobe® PhotoShop® may begin to save a file only to run out of disk space before completing the operation. Applications should verify that all necessary resources are available before beginning an operation.

6.23.1 Pattern

This is very similar to “Using Applications Concurrently.” The application must verify that necessary resources are available from a resource manager and inform the user if sufficient resources are not available to complete a task.

6.23.2 Allocation to mechanism hierarchy

Separation Encapsulation of function

The resource manager is encapsulated apart from the rest of the system.

Models System

The resource manager must maintain a model of the system to know which resources to verify and to determine a plan of action in the event that needed resources are unavailable.

6.24 Operating Consistently Across Views

A user may become confused by functional deviations between different views of the same data. Commands that had been available in one view may become unavailable in another or may require different access methods. For example, users cannot run a spell check in the Outline View utility found in a mid-90’s version of Microsoft Word. Systems should make

commands available based on the type and content of a user's data, rather than the current view of that data, as long as those operations make sense in the current view.

For example, allowing users to perform operations on individual points in a scatter plot while viewing the plot at such a magnification that individual points cannot be visually distinguished does not make sense. A naïve user is likely to destroy the underlying data. The system should prevent selection of single points when their density exceeds the resolution of the screen, and inform the user how to zoom in, access the data in a more detailed view, or otherwise act on single data points.

(See: Providing Good Help and Supporting Visualization)

6.24.1 Pattern

Figure 17 shows the relevant components that enable consistent operation across views. The data that is being viewed should be separated from the view description. The presentation maps the data through the description of the user's requested view. Commands are also separated from the data. This allows the commands to operate on the data without knowing the view of the data requested by the user.

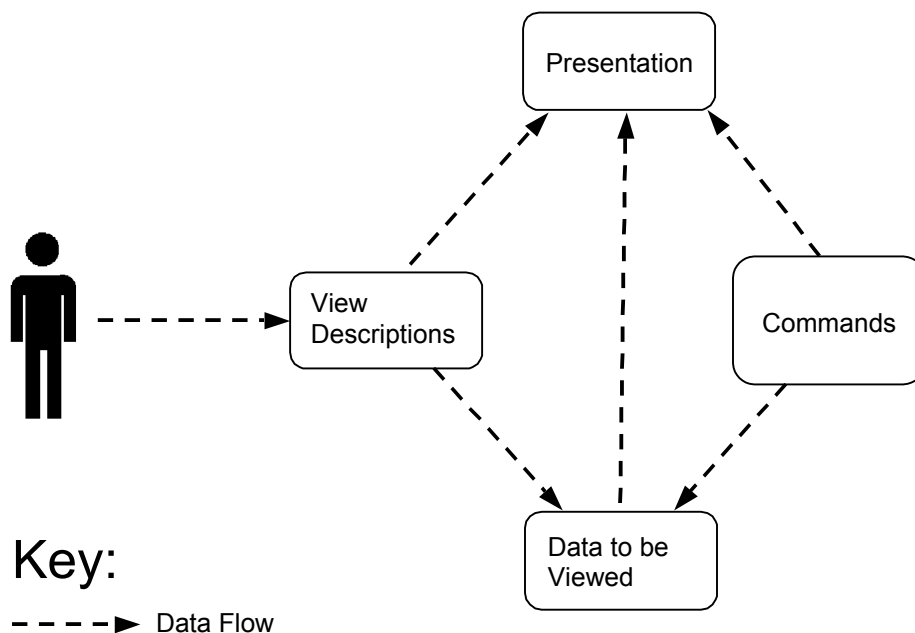


Figure 17. Consistent Operation

6.24.2 Allocation to mechanism hierarchy

Separation

Data from Commands

In systems in which users select the current view from a toolbar, the view change commands are separated from the view description.

Separation

Data from View

When the view description is kept separate from the viewed data, commands that manipulate the data can be executed without referencing the current view.

6.25 Making Views Accessible

A user may want to see data from another point of view. For example, a user may wish to see the outline of a long document and the details of the prose. If certain views become unavailable in certain modes of operation, or if switching between views is cumbersome, the user's ability to gain insight through multiple perspectives will be constrained.

6.25.1 Pattern

Figure 17 (Consistent Operation) also satisfies this scenario. Switching between views is a matter of changing the active view description and should be independent of the mode or of what is currently on the screen.

6.25.2 Allocation to mechanism hierarchy

Separation

Data from Commands

In systems where users select the current view from a toolbar, the view change commands are separated from the view description.

Separation

Data from View

When the view description is kept separate from the viewed data, the system can change views without referencing the current view.

6.26 Supporting Visualization

A user wishes to see data from a different viewpoint. Systems should provide a reasonable set of task-related views to enhance users' ability to gain additional insight while solving problems. For example, Microsoft Word provides several views to help users compose documents, including Outline and Page Layout modes.

6.26.1 Pattern

Figure 17 (Consistent Operation) also satisfies this scenario. Task-related views can be described in the view description and adding views should be an easy set of modifications. Switching between views then is a matter of changing the active view description and should be independent of the mode or of what is currently on the screen.

6.26.2 Allocation to mechanism hierarchy

Separation

Data from Commands

In systems where users select the current view from a toolbar, the view change commands are separated from the view description.

Separation

Data from View

Each view of the data exists independently from the data itself.

7 Cross-Referencing Benefits and Mechanisms

In this chapter, we present a matrix (Figure 18) that puts the mechanism hierarchy on one axis and the benefit hierarchy on the other. Each cell contains the general usability scenarios that correspond to the mechanism and benefit hierarchies.

On the one hand, this matrix reproduces the information presented in Sections 4 and 6. On the other, the matrix provides additional benefits. The software design team can decide which usability benefits are most valued in a particular project, use the matrix to focus on the general scenarios providing those benefits to see which are applicable to the project, and then read off the architectural mechanisms necessary to implement those scenarios. The team can use this information to generate the architecture or to evaluate an existing architecture to see what usability risks might be inherent in their design. Alternatively, the team could look at the mechanisms included in a current system design and use the matrix to discover which general usability scenarios could be implemented using those mechanisms, and which additional usability scenarios could be addressed with only small changes to the architecture. We expect this matrix to be the vehicle for referencing the work presented here and thereby increase its utility beyond the linear format of prose and diagrams.

Figure 18. Benefit and Mechanism Matrix

Architectural Mechanisms		Increases individual effectiveness							Reduces impact of system errors		Increases confidence and comfort
		Expedites routine performance		Improves non-routine performance		Reduces impact of mistakes		Tolerates system errors	Prevents system errors		
		Accelerates error-free portion	Reduces impact of slips	Supports problem-solving	Facilitates learning	Prevents mistakes	Accommodates mistakes				
Separation	Encapsulation of function	4, 13, 14, 15, 20, 23	3, 4	3, 4, 13, 20	4, 13, 20	4, 13, 20	3, 9, 14	23	3, 23		
	Data from the view of that data	12, 13, 24, 25	12	12, 13, 22, 24, 25, 26	12, 13, 24	12, 13, 22, 24	12			12	
	Data from commands	1, 24, 25	5, 17	5, 17, 24, 25, 26	5, 17, 24	1, 5, 17, 24	1, 5, 17			17	
	Authoring from execution	1, 2	2			1, 2	1, 2				
Replication	Data	16									
	Commands	2	2	22		2, 22	2				
Indirection	Data	7, 11, 14	11	7, 11			14				
	Function	6, 14, 20		6, 20	20	20	14	6	6		
Recording		2, 7	2, 3, 21	3, 7, 21		2	2, 3, 21	3, 8	3, 6		
Preemptive Scheduling		15, 18, 19	2, 3, 5, 17, 18	3, 5, 10, 17	5, 10, 17	5, 17, 19	3, 5, 17	3	3	17, 18	
Models	Task	18, 19	5, 17, 18	5, 10, 17	5, 10, 17	5, 17, 19	5, 17			17, 18	
	User	12, 13, 18	5, 12, 17, 18	5, 10, 12, 13, 17, 22	5, 10, 12, 13, 17	5, 12, 13, 17, 22	5, 12, 17			12, 17, 18	
	System	4, 6, 19, 23	3, 4, 5, 17	3, 4, 5, 6, 17	4, 5, 17	3, 4, 5, 17, 19	3, 5, 17	3, 6, 23	3, 6, 23	17	

KEY

- | | | |
|--|---|---|
| 1 Aggregating data | 10 Providing good help | 19 Predicting task duration |
| 2 Aggregating commands | 11 Reusing information | 20 Supporting comprehensive searching |
| 3 Canceling commands | 12 Supporting international use | 21 Supporting Undo |
| 4 Using applications concurrently | 13 Levering Human Knowledge | 22 Working in an unfamiliar context |
| 5 Checking for correctness | 14 Modifying interfaces | 23 Verifying resources |
| 6 Maintaining device independence | 15 Supporting multiple activity | 24 Operating consistently across views |
| 7 Evaluating the system | 16 Navigating within a single view | 25 Making views accessible |
| 8 Recovering from failure | 17 Observing system state | 26 Supporting visualization |
| 9 Retrieving forgotten passwords | 18 Working at the user's pace | |

8 Further Work

This work is by no means complete. There are a number of activities that remain to be done to simplify the support of usability at the software architectural design stage of a system. These activities include

- validating and extending the list of general usability scenarios. We explicitly limited our attention to single-user desktop systems. The scenarios should be extended to include mobile and multi-user environments. Although many of the general scenarios presented here will still be applicable, these environments will introduce their own additional usability requirements. Furthermore, the scenarios need to be validated in practice. Their utility still must be determined, although we have used them successfully in several evaluation and design exercises to date.
- understanding the effect on other attributes of usability architecture patterns. In order for an architect to adopt a particular architectural pattern to support usability, the effect of this pattern on other quality attributes such as performance, modifiability, security and reliability should be understood. There is ongoing work at the SEI in this area [Bass 2000].
- understanding the usability impact of architectural mechanisms used to achieve other attributes. For example, what is the usability impact of a firewall? Again, there is ongoing work at the SEI in this area [Bass 2000].

References/Bibliography

- Bass 1998** Bass, L.; Clements, P. & Kazman, R. *Software Architecture in Practice*. Reading, MA: Addison Wesley Longman, 1998.
- Bass 2000** Klein, M. & Bachmann, F. *Quality Attribute Design Primitives* (CMU/SEI-2000-TN-2000-017). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000.
- Bhavnani 2000** Bhavnani, S. K. & John, B. E. "The Efficient Use of Complex Computer Systems." *Human-Computer Interaction* 15, 2: 107-137.
- Card 1983** Card, S. K.; Moran, T. P. & Newell, A. *The Psychology of Human-Computer Interaction*. Hillsdale, NJ: Erlbaum, 1983.
- Gram 1996** Gram, C. & Cockton, G. *Design Principles for Interactive Systems*. London, England: Chapman and Hall, 1996.
- Newell 1972** Newell, A. & Simon, H. A. *Human Problem Solving*. Englewood Cliffs, NJ: Prentice-Hall, 1972.
- Newman 1995** Newman, W. & Lamming, M. *Interactive System Design*. Wokingham, England: Addison-Wesley Publishing, 1995.
- Nielsen 1993** Nielsen, J. *Usability Engineering*. Boston, MA: Academic Press Inc., 1993.
- Shneiderman 1998** Shneiderman, B. *Designing the User Interface*, 3rd ed. Reading, MA: Addison-Wesley, 1998.
- Beyer 1998** Beyer, H. & Holtzblatt, K. *Contextual Design*. San Francisco, CA: Morgan Kaufmann Publishers, Inc., 1998.

Miller 1987

Miller, D. P. & Swain, A. D. "Human Error and Human Reliability," 219-257. *Handbook of Human Factors*, ed. Gavriel Salvendy. New York: John Wiley and Sons, Inc.

REPORT DOCUMENTATION PAGE			<i>Form Approved OMB No. 0704-0188</i>
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.			
1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE		3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE Achieving Usability Through Software Architecture		5. FUNDING NUMBERS F19628-00-C-0003	
6. AUTHOR(S) Len Bass, Bonnie E. John, Jessie Kates			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213		8. PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-2001-TR-005	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116		10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES			
12A DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS		12B DISTRIBUTION CODE	
13. ABSTRACT (MAXIMUM 200 WORDS) In this paper, we present an approach to improving the usability of software systems by means of software architectural decisions. We identify specific connections between aspects of usability, such as the ability to "undo," and software architecture. We also formulate each aspect of usability as a scenario with a characteristic stimulus and response. For every scenario, we provide an architecture pattern that implements its aspect of usability. We then organize the usability scenarios by category. One category presents the benefits of these aspects of usability to users or their organizations. A second category presents the architecture mechanisms that directly relate to the aspects of usability. Finally, we present a matrix that correlates these two categories with the general scenarios that apply to them.			
14. SUBJECT TERMS Software architecture, software systems, usability, general scenarios, usability evaluators, architecture patterns, architecture mechanisms, architecture design		15. NUMBER OF PAGES 102	
16. PRICE CODE			
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18 298-102