Avoiding “We can’t change that!”: Software Architecture & Usability

Bonnie E. John
Human-Computer Interaction Institute
bej@cs.cmu.edu

Len Bass
Software Engineering Institute
ljb@sei.cmu.edu

Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh PA 15123

CHI 2003 Tutorial

Table of Contents

Agenda

Time	Topic
6:00-6:15	Instructor introduction, audience background & tutorial objectives
6:15-6:35	The causes of “We can’t change that!”
6:35-6:55	Known solutions for certain types of usability changes
6:55-7:15	Usability & Software Architecture Approach (U&SA)
7:15-7:45	BREAK
7:45-8:10	Example: Canceling commands
8:10-8:25	Example: Reusing information
8:25-8:40	Example: Supporting international use
8:40-8:55	Example: Observing system state
8:55-9:20	U&SA in analysis and design
9:20-9:30	Wrap-up

Instructor Biographies

Bonnie John is an engineer (B.Engr., The Cooper Union, 1977; M. Engr. Stanford, 1978) and cognitive psychologist (M.S. Carnegie Mellon, 1984; Ph. D. Carnegie Mellon, 1988) who has worked both in industry (Bell Laboratories, 1977-1983) and academe (Carnegie Mellon University, 1988-present). She is an Associate Professor in the Human-Computer Interaction Institute and the Director of the Masters Program in HCI. Her research includes human performance modeling, usability evaluation methods, and the relationship between usability and software architecture. She consults for many industrial and government organizations.

Len Bass is an expert in software architecture & architecture design methods. Author of six books including two textbooks on software architecture & UI development, Len consults on large-scale software projects in his role as Senior MTS on the Architecture Trade-off Analysis Initiative at the Software Engineering Institute. His research area is the achievement of various software quality attributes through software architecture and he is the developer of software architecture analysis and design methods. Len is also the past chair of the International Federation of Information Processing Working Group on User Interface Engineering.

Objectives of the course

Participants in this tutorial will

- Understand basic principles of software architecture for interactive systems and its relationship to the usability of that system

- Be able to evaluate whether common usability scenarios will arise in the systems they are developing and what implications these usability scenarios have for software architecture design

- Understand patterns of software architecture that facilitate usability, and recognize architectural decisions that preclude usability of the end-product, so that they can effectively bring usability considerations into early architectural design.

Abstract

The usability analyses or user test data are in; the development team is poised to respond. The software had been carefully modularized so that modifications to the UI would be fast and easy. When the usability problems are presented, someone around the table exclaims, “Oh, no, we can’t change *THAT*!” The requested modification or feature reaches too far in to the architecture of the system to allow economically viable and timely changes to be made. Even when the functionality is right, even when the UI is separated from that functionality, architectural decisions made early in development have precluded the implementation of a usable system. The members of the design team are frustrated and disappointed that despite their best efforts, despite following current best practice, they must ship a product that is far less useable than they know it could be.

This scenario need not be played out if usability concerns are considered during the earliest design decisions of a system, that is, during the architectural design, just as concerns for performance, availability, security, modifiability, and other quality attributes are considered. The relationships between these attributes and architectural decisions are relatively well understood and taught routinely in software architecture courses. However, the prevailing wisdom in the last 20 years has been that usability had no architectural role except through modifiability; design the UI to be easily modified and usability will be realized through iterative design, analysis and testing. Separation of the user interface has been quite effective, and is commonly used in practice, but it has problems. First, there are many aspects of usability that require architectural support other than separation, and, second, the later changes are made to the system, the more expensive they are to achieve. Forcing usability to be achieved through modification means that time and budget pressures are likely to cut off iterations on the user interface and result in a system that is not as usable as possible.

Recent developments made jointly by this tutorial’s instructors at the Software Engineering and Human-Computer Interaction Institutes at Carnegie Mellon University have established the relationship between architectural decisions and usability. This tutorial will teach this relationship. It will give usability specialists and software developers alike an explicit link between their two realms of expertise, allowing both to participate more effectively in the early design decisions of an interactive system. It will give the entire design team the tools to consider usability from the very earliest stages of design, and allow informed architectural decisions that do no preclude usability.

The scene

The usability analyses or user test data are in; the development team is poised to respond. The software had been carefully modularized so that modifications to the UI would be fast and easy. When the usability problems are presented, a developer around the table exclaims, “Oh, no, we can’t change THAT!”

The scene

The usability analyses or user test data are in; the development team is poised to respond. The software had been carefully modularized so that modifications to the UI would be fast and easy. When the usability problems are presented, a developer around the table exclaims, “Oh, no, we can’t change THAT!”

The requested modification, feature, functionality, reaches too far in to the architecture of the system to allow economically viable and timely changes to be made.

- **Even when the functionality is right,**
- **Even when the UI is separated from that functionality,**
- **Architectural decisions made early in development can preclude the implementation of a usable system.**

Outline of tutorial

Analyze the causes of the “We can’t change THAT” problem

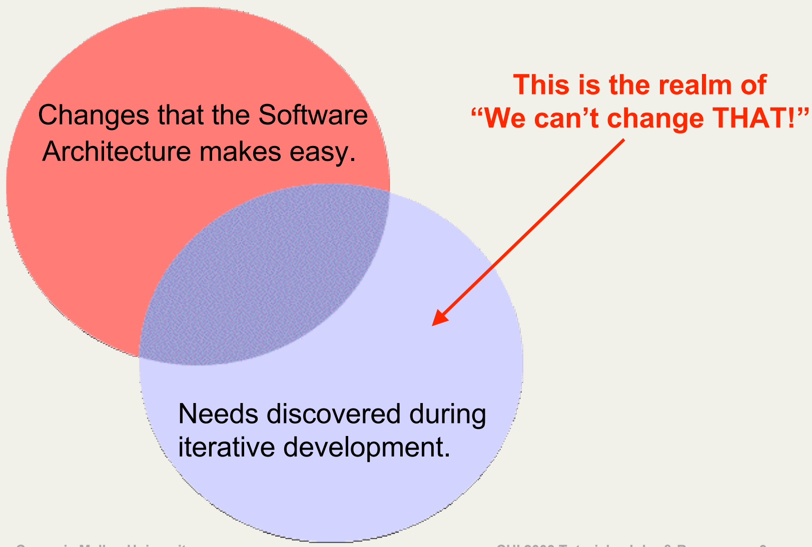
Discuss known solutions for certain classes of usability changes and why they don’t work for everything.

Usability & Software Architecture (U&SA)
- General usability scenarios with architectural impact
- Architectural patterns and tactics to support usability

Applying U&SA to architecture evaluation and design

Carnegie Mellon

What leads to “We can’t change THAT!”?
So what is the cause?



What is software architecture?

Software architecture is the high-level structural design

- Enumeration of all major modules
- Enumeration of responsibilities for each module
- Interaction among modules specified
  - Control and data flow
  - Sequencing information
  - Protocols of interaction
  - Allocation to hardware

Software architecture is the first system artifact that can be analyzed with respect to the quality attributes important to the particular system

Carnegie Mellon

U&SA's Strategy – 2

Identify those aspects of usability that are "architecturally sensitive" and embody them in small scenarios

Provide checklist of important software responsibilities, software tactics, and possible architecture patterns to satisfy these scenarios

Integrate architecturally sensitive aspects of usability into software architecture evaluation methods (e.g., ATAM^{SM})

Use architecture patterns within software architecture generation methods (e.g., ADD)

Short descriptions of the Attribute Tradeoff Analysis Method^{SM} (ATAM^{SM}) and Attribute-Driven Design (ADD) can be found in Bass, L.; Clements, P. & Kazman, R. (2003). *Software Architecture in Practice, 2nd edition*. Reading, MA: Addison Wesley Longman.

Carnegie Mellon

What does architecturally-sensitive mean?

A scenario is architecturally-sensitive if it is difficult to support by patterns that only separate the user interface from the application.

Solution may:
- Require that multiple modules interact in particular ways
- Require that related information and actions be placed in a single module and therefore can be easily changed

Consider the previously mentioned examples in J2EE/MVC:
- Changing color of font modifies only View
  - NOT architecturally-sensitive
- Changing color of font modifies only Controller
  - NOT architecturally-sensitive
- Adding a cancel command modifies all modules
  - IS architecturally-sensitive

Architecturally-sensitive usability scenarios

Focussed on both end users and developers

Each usability scenario is a short description of an interaction with a system.

Initially focused on single user at a desktop, but have also proven useful in co-located collaborative environments.

Currently 27 scenarios (see Appendix I), e.g.,

- cancellation
- information reuse (not having to enter same information multiple times)
- observing system state

Elements of an architecturally-sensitive usability scenario package

General usability scenario

Usability benefits to the user

Checklist of responsibilities to allocate at architecture design time

Example architectural pattern based on J2EE/MVC
- Software tactics to implement the pattern

Usability benefits hierarchy

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

Carnegie Mellon

Software architecture tactics hierarchy

Localize modifications
- Hide information
- Separate data from commands
- Separate data from the view of that data
- Separate authoring from execution

Maintain multiple copies
- Data
- Commands

Use an intermediary
- Data
- Function

Recording

Preemptive scheduling policy

Support system initiative
- Task model
- User model
- System model

Examples of using general scenario packages

Demonstrate how to use scenario packages

- Canceling commands
- Reusing information
- Supporting international use
- Observing system state

Applying the scenarios

Scenarios should be used as a checklist during the requirements process.

Scenarios are revisited during the design process to make sure they are supported by the architecture.

Scenarios act as a checklist for developers to ensure they are implemented in the source code.

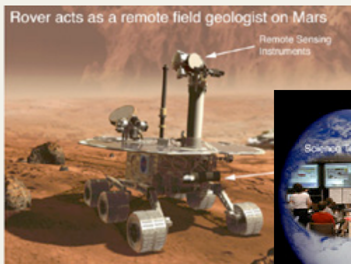
Scenarios should be re-checked during any modification effort

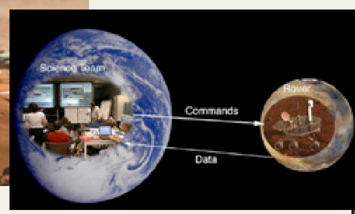
Scenarios can also come into procurement decisions, for example, supporting international use may require purchasing a database that supports double-byte characters for storing text with non-roman lettering.

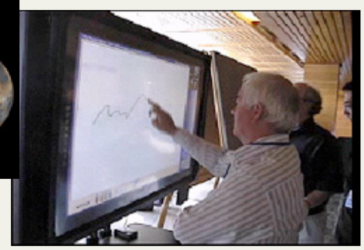
Carnegie Mellon

U&SA applied to the NASA MERBoard

The Mars Exploration Rover Board (MERBoard) is a collaborative workspace to aid engineers and scientists analyze data and plan the work of the Mars Exploration Rover







Pictures from the IBM website
http://www.research.ibm.com/resources/news/20020603_merboard.shtml

Applicability of the scenarios

Design & development team found 25 of 27 scenarios to be applicable to their project

17 of the 25 applicable scenarios needed by the next field trial; 8 were for the longer term

Easy for the development team to give concrete examples of these scenarios for their users, often from direct observation during the field trials

Summary of applying the U&SA approach to MERBoard

Scenarios were well received by the developers, readily understood how they fit (or didn't) to their system

Scenarios *DID* apply to collaborative workspace

- We don't know if there will be collaborative-specific scenarios yet

Scenarios *HAD* an impact on the architecture redesign

Process did not seem too onerous

Carnegie Mellon

Benefit/Tactic Matrix

Architectural Tactics ↓ / Usability Benefits →		Increases individual effectiveness						Reduces impact of system errors		Increases confidence and comfort
		Expedites routine performance		Improves non-routine performance		Reduces impact of mistakes				
		Accelerates error-free portion	Reduces impact of slips	Supports problem-solving	Facilitates learning	Prevents mistakes	Accommodates mistakes	Tolerates system errors	Prevents system errors	
Localize Modifications	Hide information	4, 13, 14, 15, 20, 23		4, 13, 20	4, 13, 20	4, 13, 20	9, 14		23	
	Separate 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
	Separate data from commands	1, 24, 25	5, 17	5, 17, 24, 25, 26	5, 17, 24	1, 5, 17, 24	1, 5, 17			17
	Separate authoring from execution	1, 2	2			1, 2	1, 2			
Maintain multiple copies	Data	16								
	Commands	2	2	22		2, 22	2			
Use an intermediary	Data	7, 11, 14	11	7, 11			14			
	Function	6, 14, 20, 27	27	6, 20	20	20, 27	14		6	27
Recording		2, 7	2, 3, 21	3, 7, 21		2	2, 3, 21	3, 8		
Preemptive scheduling policy		15, 18, 19	3, 5, 17, 18	3, 5, 10, 17	5, 10, 17	5, 17, 19	3, 5, 17	3		17, 18
Support system initiative	Task model	18, 19	5, 17, 18	5, 10, 17	5, 10, 17	5, 17, 19	5, 17			17, 18
	User model	12, 18	5, 12, 17, 18	5, 10, 12, 17, 22	5, 10, 12, 17	5, 12, 17, 22	5, 12, 17			12, 17, 18
	System model	4, 6, 19, 23	3, 5, 17	3, 4, 5, 6, 17	4, 5, 17	4, 5, 17, 19	3, 5, 17	3	6, 23	17

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

A larger version of this matrix appears in Appendix IV.