Avoiding "We can't change *<u>THAT!</u>*": Software Architecture & Usability

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

Len Bass
Software Engineering Institute
Carnegie Mellon University
ljb@sei.cmu.edu

Natalia Juristo
School of Computing
Technical University of Madrid
natalia@fi.upm.es

Maribel Sanchez-Segura
Computer Science Department
Carlos III University of Madrid
Mariaisabel.sanchez@uc3m.es

CHI2004 Tutorial

Table of Contents

Agenda

Time	Topic
18:00-18:15	Instructor introduction, audience background & tutorial objectives
18:15-18:30	What is Software Architecture & What is Usability? • Basic Concepts of each
18:30-18:45	How software architecture and usability techniques fit into a software development activities
18:45-19:15	• Separation based architectural patterns and their motivation (e.g.,J2EE-MVC) • Why separation is inadequate for interactive systems
19:15-19:30	Usability-Supporting Architectural Patterns (USAP) • Introduction to the concept of a USAP
19:30-20:00	BREAK
20:00-20:30	Usability-Supporting Architectural Patterns (USAP) • The parts of a USAP • Example of a USAP - problem statement, forces, and general responsibilities
20:30-21:00	Small Group Exercise: • Applying the problem statement and forces to a real-world problem brought by the members of the breakout group (one group per instructor) • Report-out to the entire group
21:00-21:30	• Sample solution for example problem. • Using USAPs in development • Tutorial summary

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.

Dr. Natalia Juristo is a professor of software engineering with the Computing School at the Universidad Politecnica de Madrid and former Director of the MSc in Software Engineering. Dr. Juristo has a B.S. and a Ph.D. in Computing. She was fellow of the European Centre for Nuclear Research (CERN) in Switzerland in 1988, and staff of the European Space Agency (ESA) in Italy in 1989 and 1990. During 1992 she was a resident affiliate of the Software Engineering Institute at Carnegie Mellon University. She was program chair for SEKE97 and general chair for SEKE01 and SNPD02. Prof. Juristo has been the keynote speaker for CSEET03. She has been the guest editor of special issues in several journals, and a member of several editorial boards, including IEEE Software and the Journal of Empirical Software Engineering. She is a senior member of IEEE.

Maribel Sanchez-Segura has been a faculty member of the Computer Science Department in the Carlos III Technical University of Madrid since 1998. Her research interests include software engineering, interactive systems, and usability. Maribel holds a B.S. in Computer Science, a M.S. in Software Engineering and a Ph.D. in Computer Science from the Technical University of Madrid.

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.

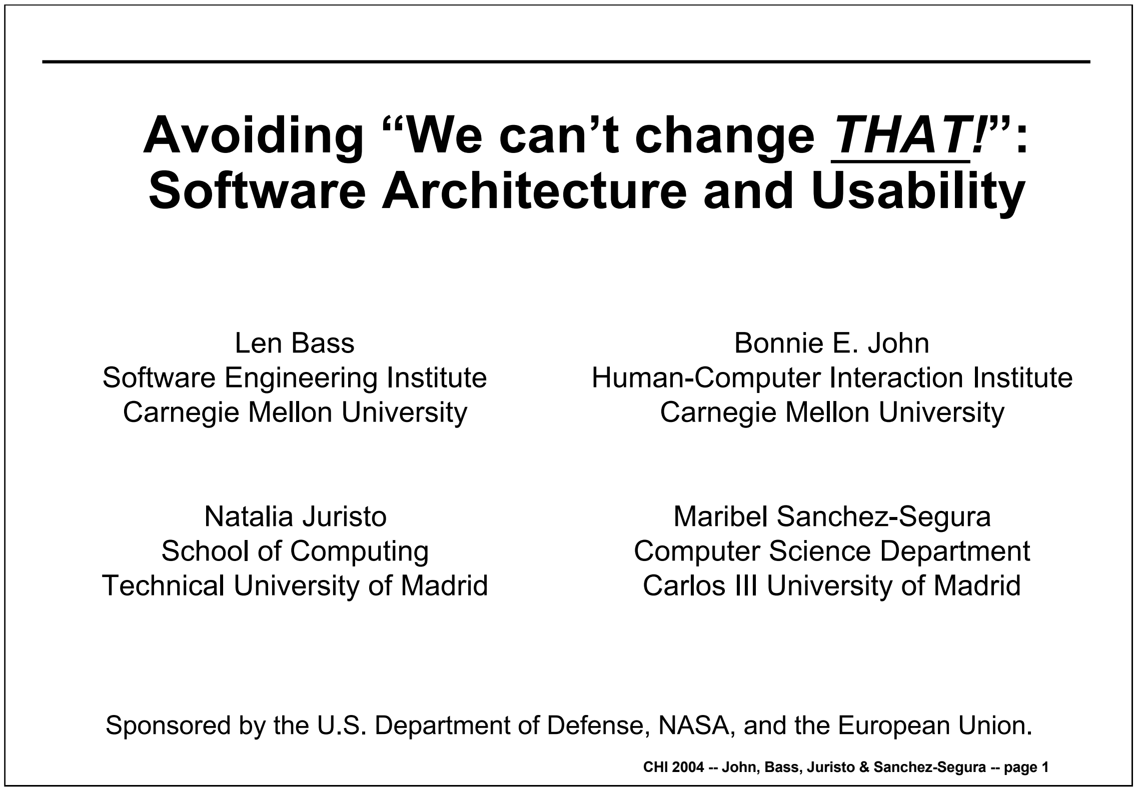
Abstract

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, 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 that are difficult to change 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 in CS curricula. 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.

Work conducted by this tutorial’s instructors at the Software Engineering and Human-Computer Interaction Institutes at Carnegie Mellon University, the Technical University of Madrid, and Carlos III University of Madrid has investigated 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.



Avoiding “We can’t change *THAT!*”: Software Architecture and Usability

Len Bass
Software Engineering Institute
Carnegie Mellon University

Bonnie E. John
Human-Computer Interaction Institute
Carnegie Mellon University

Natalia Juristo
School of Computing
Technical University of Madrid

Maribel Sanchez-Segura
Computer Science Department
Carlos III University of Madrid

Sponsored by the U.S. Department of Defense, NASA, and the European Union.

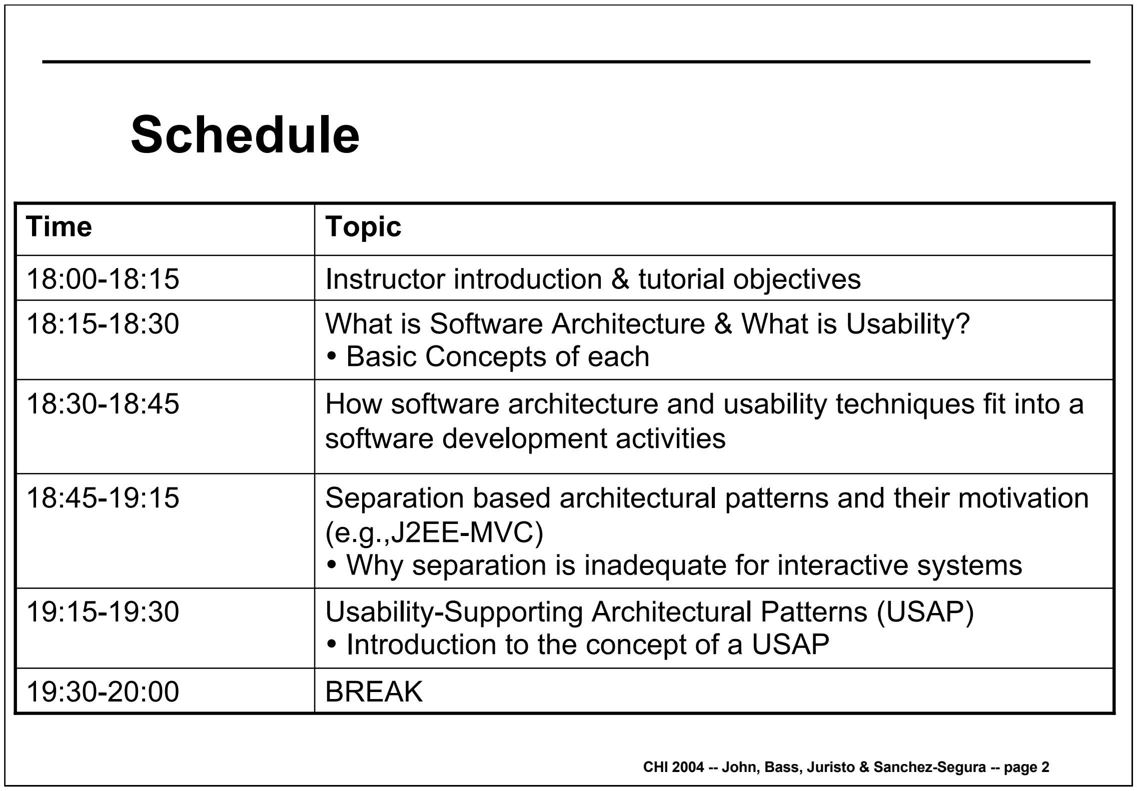
CHI 2004 -- John, Bass, Juristo & Sanchez-Segura -- page 1

Bonnie E. John
Human-Computer Interaction Institute
Carnegie Mellon University
Pittsburgh PA 15213
USA
1-412-268-7182
bej@cs.cmu.edu

Len Bass
Software Engineering Institute
Carnegie Mellon University
Pittsburgh PA 15213
USA
1-412-268-6763
ljb@sei.cmu.edu

Natalia Juristo
School of Computing
Technical University of Madrid
Campus de Montegancedo s/n
28660 Boadilla del Monte
Spain
34-91-3366922
natalia@fi.upm.es

Maribel Sanchez-Segura
Computer Science Department
Universidad Carlos III de Madrid
Avda. de la Universidad, 30
28911 Leganes
Spain
34-91-6249421
Mariaisabel.sanchez@uc3m.es



Schedule

Time	Topic
18:00-18:15	Instructor introduction & tutorial objectives
18:15-18:30	What is Software Architecture & What is Usability? • Basic Concepts of each
18:30-18:45	How software architecture and usability techniques fit into a software development activities
18:45-19:15	Separation based architectural patterns and their motivation (e.g.,J2EE-MVC) • Why separation is inadequate for interactive systems
19:15-19:30	Usability-Supporting Architectural Patterns (USAP) • Introduction to the concept of a USAP
19:30-20:00	BREAK

Schedule

Time	Topic
20:00-20:30	Usability-Supporting Architectural Patterns (USAP) • The parts of a USAP • Example of a USAP - problem statement, forces, and general responsibilities
20:30-21:00	Small Group Exercise: • Applying the problem statement and forces to a real-world problem brought by the members of the breakout group (one group per instructor) • Report-out to the entire group
21:00-21:30	Sample solution for example problem. Using USAPs in development Tutorial summary

Introductions

Who are we?

- Len Bass
- Bonnie John
- Natalia Juristo
- Maribel Sanchez-Segura

Who are you?

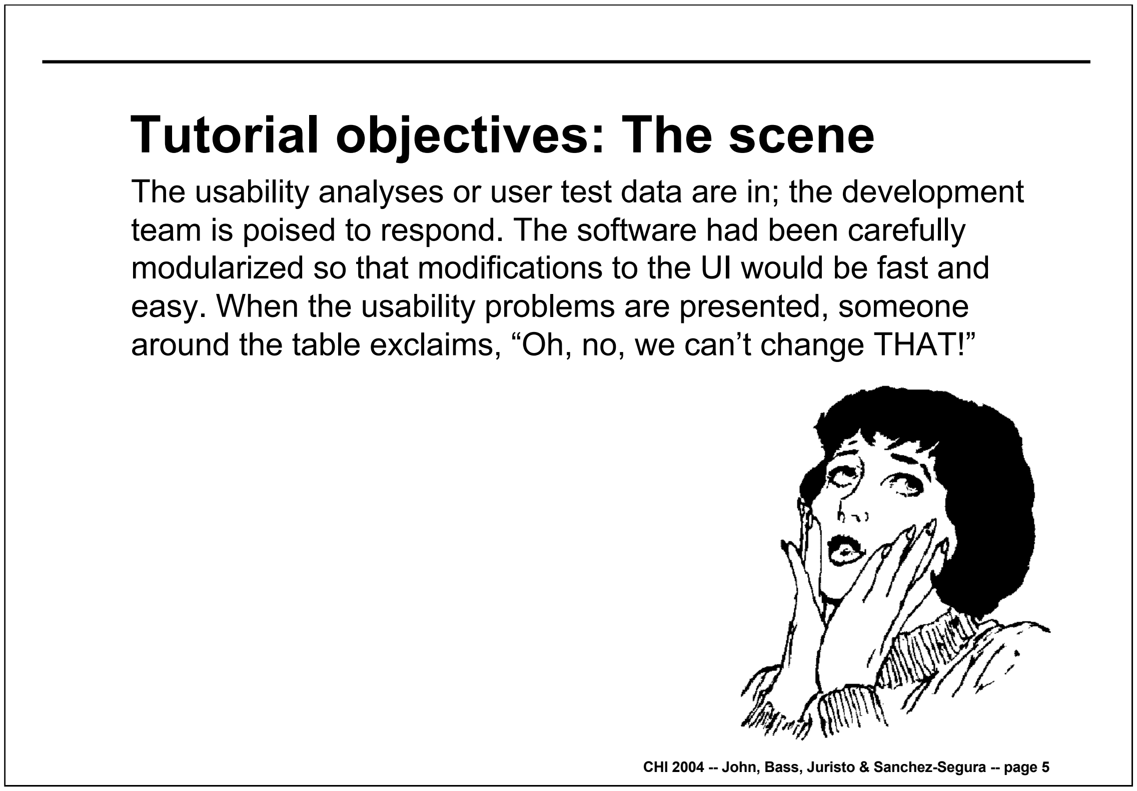
What do you want to get out of this tutorial?

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.

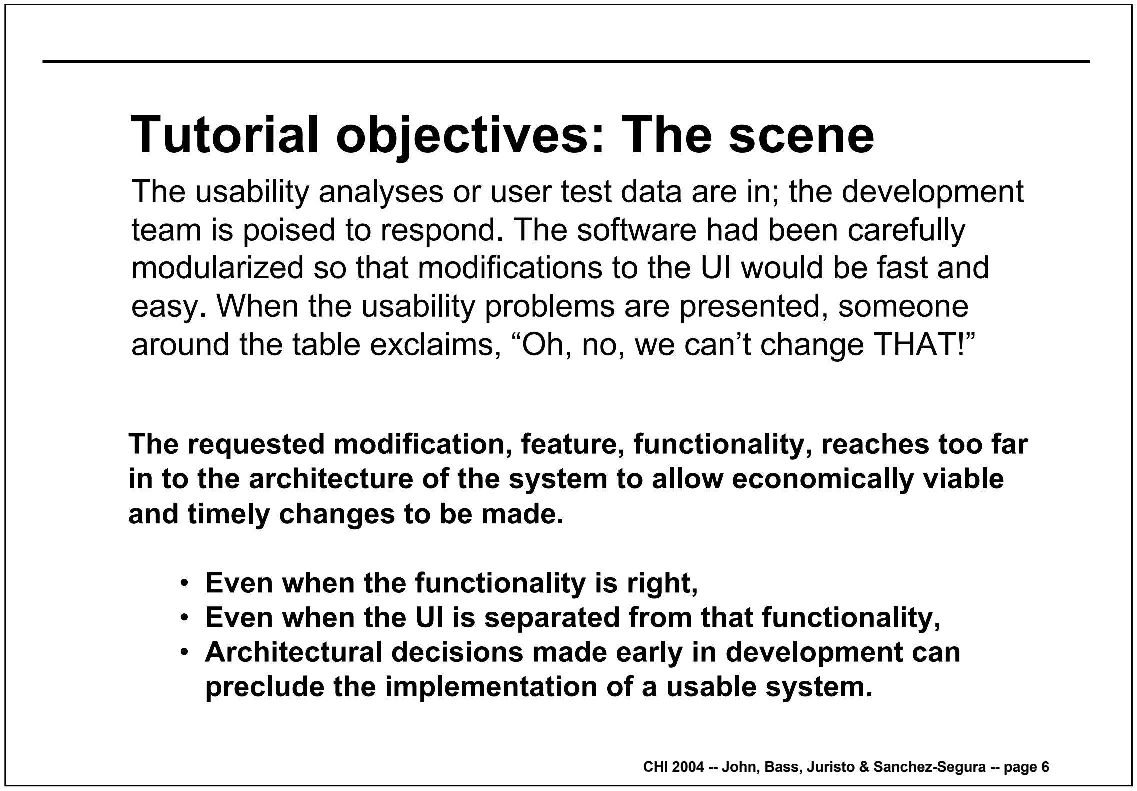
Dr. Natalia Juristo is a professor of software engineering with the Computing School at the Universidad Politecnica de Madrid. From 1992 until 2002 she was the Director of the MSc in Software Engineering. Dr. Juristo has a B.S. and a Ph.D. in Computing. She was fellow of the European Centre for Nuclear Research (CERN) in Switzerland in 1988, and staff of the European Space Agency (ESA) in Italy in 1989 and 1990. During 1992 she was a resident affiliate of the Software Engineering Institute at Carnegie Mellon University. She was program chair for SEKE97 and general chair for SEKE01 and SNPD02. Prof. Juristo has been the keynote speaker for CSEET03. She has been the guest editor of special issues in several journals, including the Journal of Software and Systems, Data and Knowledge Engineering and the International Journal of Software Engineering and Knowledge Engineering Dr. Juristo has been a member of several editorial boards, including IEEE Software and the Journal of Empirical Software Engineering. She is a senior member of IEEE.

Maribel Sanchez-Segura has been a faculty member of the Computer Science Department in the Carlos III Technical University of Madrid since 1998. Her research interests include software engineering, interactive systems, and usability. Maribel holds a B.S. in Computer Science, a M.S. in Software Engineering and a Ph.D. in Computer Science from the Technical University of Madrid.



Tutorial objectives: 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, someone around the table exclaims, “Oh, no, we can’t change THAT!”

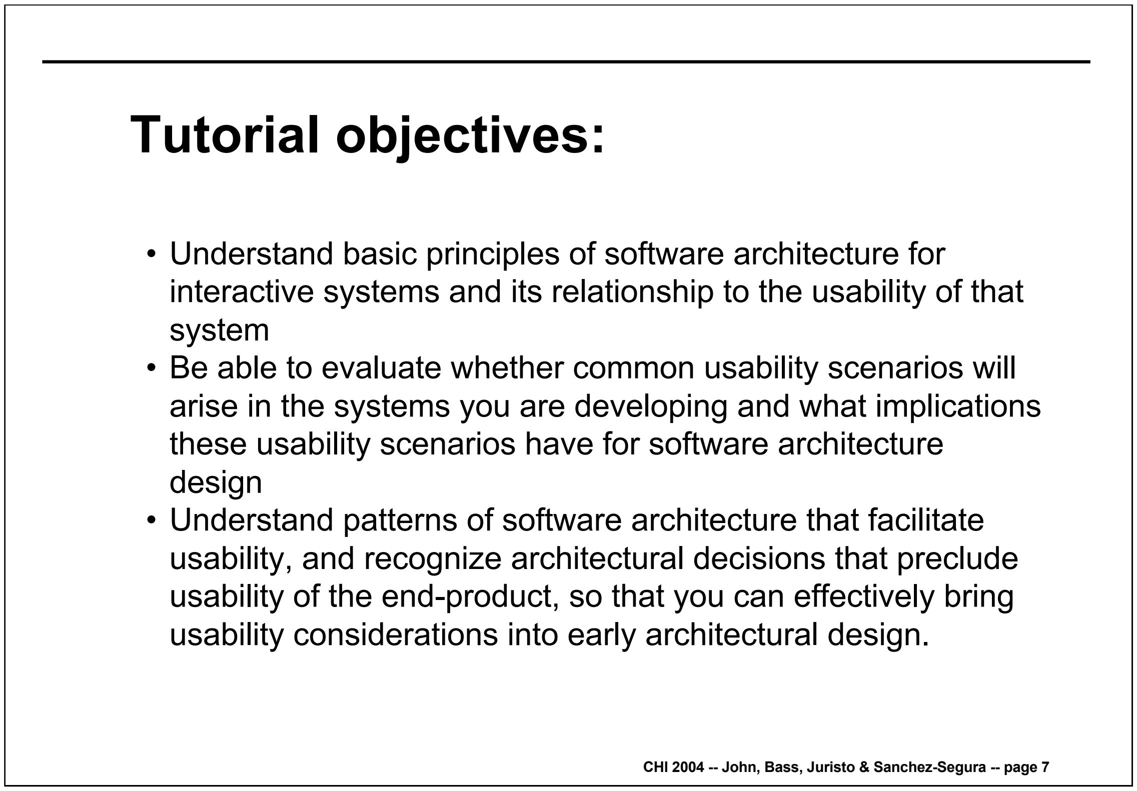

Tutorial objectives: The scene

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, someone 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.

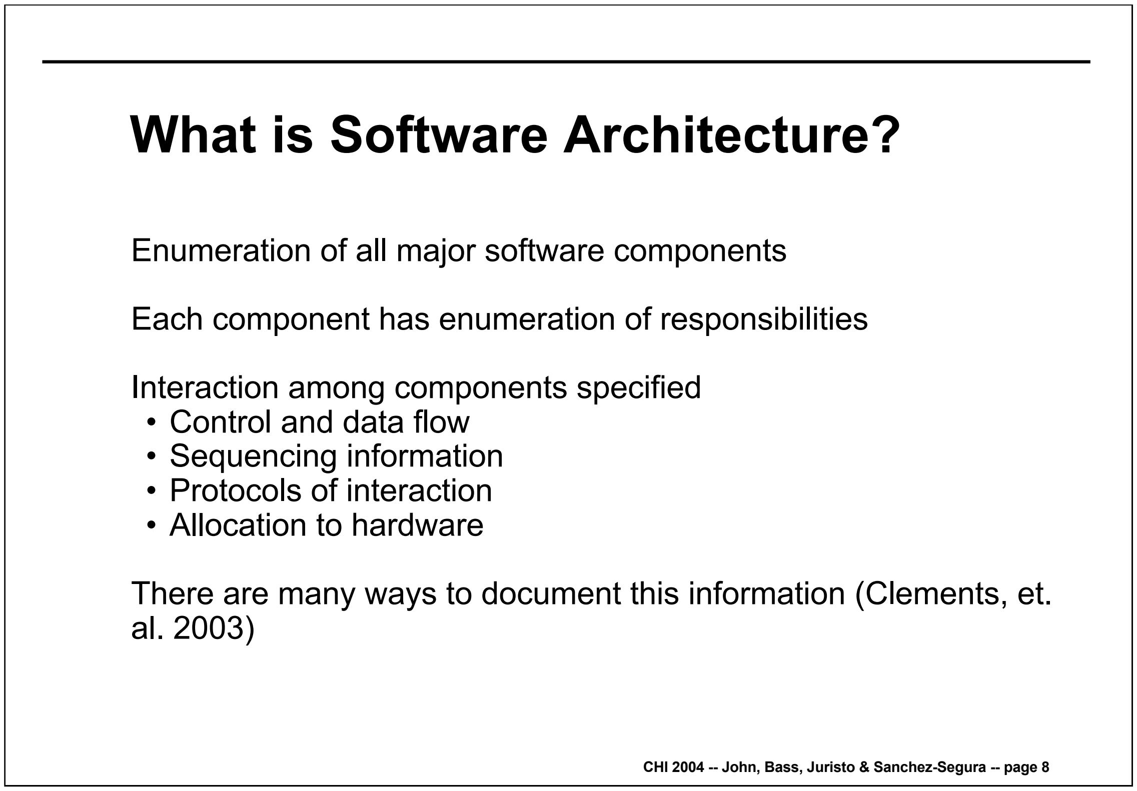
- **Even when the functionality is right,**
- **Even when the UI is separated from that functionality,**
- **Architectural decisions made early in development can preclude the implementation of a usable system.**

CHI 2004 -- John, Bass, Juristo & Sanchez-Segura -- page 6



Tutorial objectives:

- 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 you 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 you can effectively bring usability considerations into early architectural design.



What is Software Architecture?

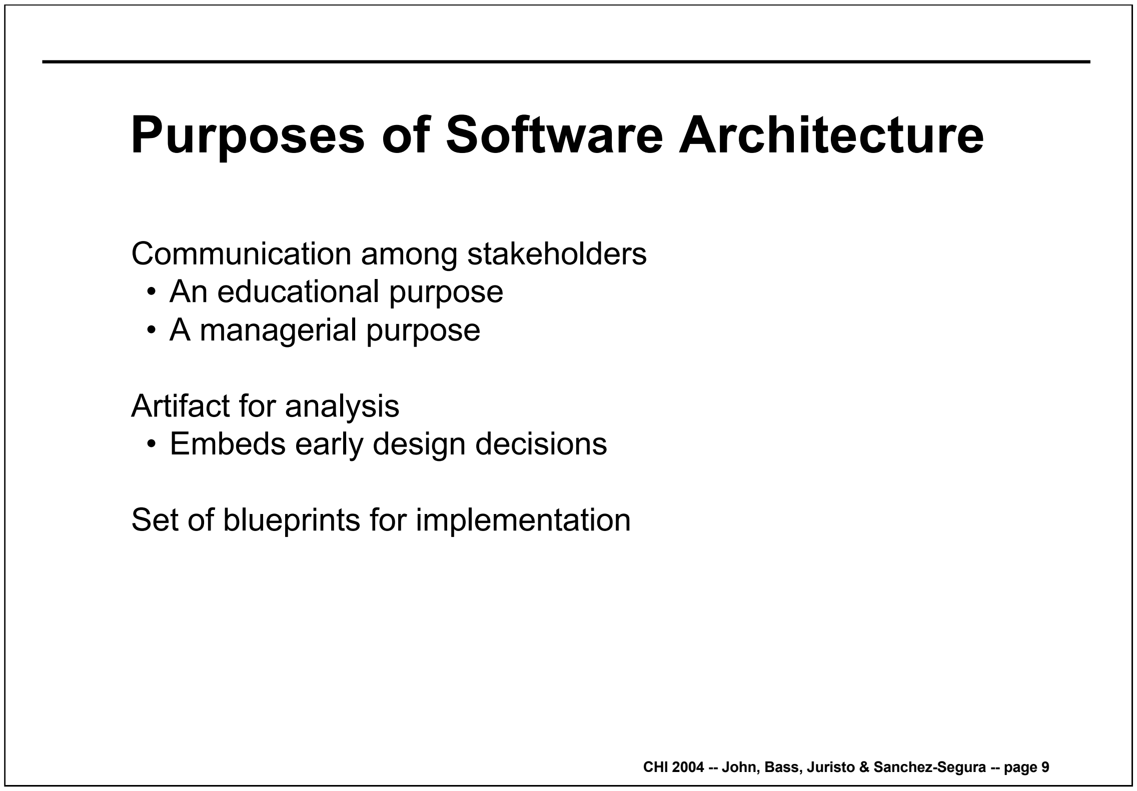
Enumeration of all major software components

Each component has enumeration of responsibilities

Interaction among components specified
- Control and data flow
- Sequencing information
- Protocols of interaction
- Allocation to hardware

There are many ways to document this information (Clements, et. al. 2003)

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., & Stafford J., (2003) *Documenting Software Architectures: Views and Beyond*, Addison Wesley.



Purposes of Software Architecture

Communication among stakeholders
- An educational purpose
- A managerial purpose

Artifact for analysis
- Embeds early design decisions

Set of blueprints for implementation

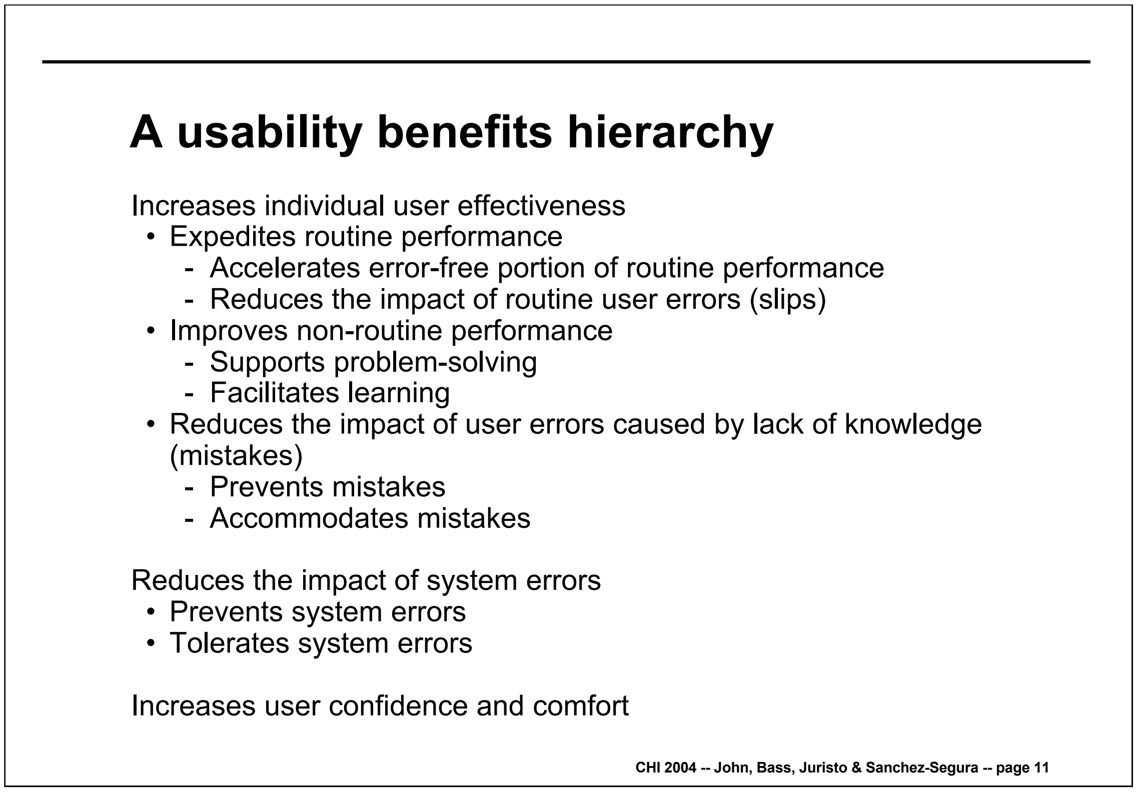
What does usability mean?

As many definitions as there are authors!

What's important depends on context of use

Some commonly-seen aspects
- efficiency of use
- time to learn to use efficiently
- support for exploration and problem-solving
- user satisfaction (e.g., trust, pleasure, acceptance by discretionary users)

Our concern is which of these can be influenced by architectural decisions

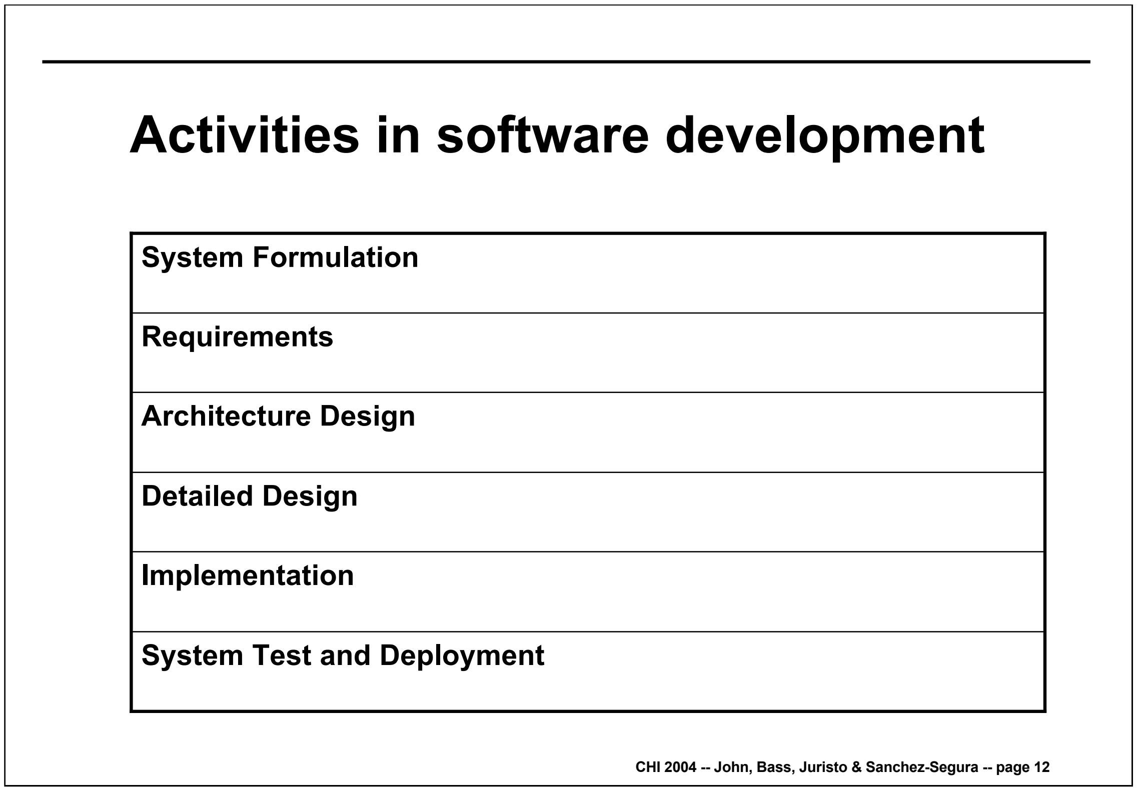


A usability benefits hierarchy

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

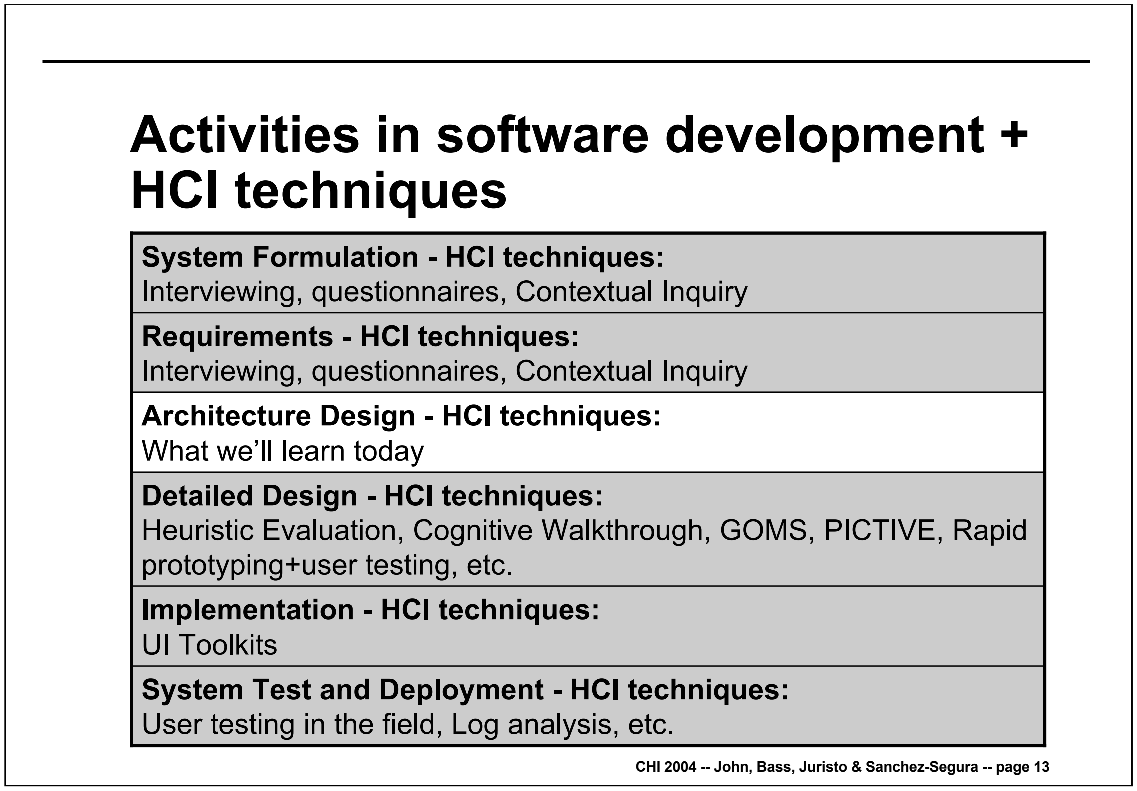
Reduces the impact of system errors
- Prevents system errors
- Tolerates system errors

Increases user confidence and comfort



Activities in software development

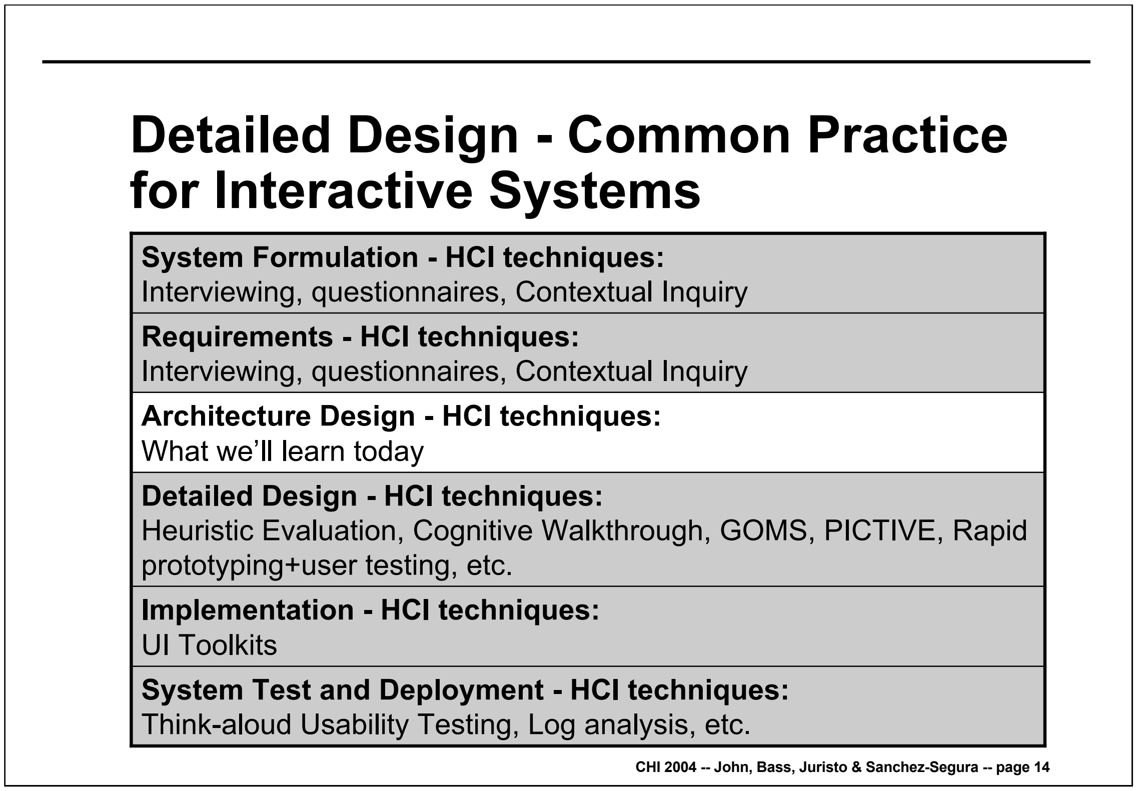
System Formulation
Requirements
Architecture Design
Detailed Design
Implementation
System Test and Deployment



Activities in software development + HCI techniques

System Formulation - HCI techniques: Interviewing, questionnaires, Contextual Inquiry
Requirements - HCI techniques: Interviewing, questionnaires, Contextual Inquiry
Architecture Design - HCI techniques: What we'll learn today
Detailed Design - HCI techniques: Heuristic Evaluation, Cognitive Walkthrough, GOMS, PICTIVE, Rapid prototyping+user testing, etc.
Implementation - HCI techniques: UI Toolkits
System Test and Deployment - HCI techniques: User testing in the field, Log analysis, etc.

CHI 2004 -- John, Bass, Juristo & Sanchez-Segura -- page 13



Detailed Design - Common Practice for Interactive Systems

System Formulation - HCI techniques: Interviewing, questionnaires, Contextual Inquiry
Requirements - HCI techniques: Interviewing, questionnaires, Contextual Inquiry
Architecture Design - HCI techniques: What we'll learn today
Detailed Design - HCI techniques: Heuristic Evaluation, Cognitive Walkthrough, GOMS, PICTIVE, Rapid prototyping+user testing, etc.
Implementation - HCI techniques: UI Toolkits
System Test and Deployment - HCI techniques: Think-aloud Usability Testing, Log analysis, etc.

Detailed Design - Common Practice for Interactive Systems

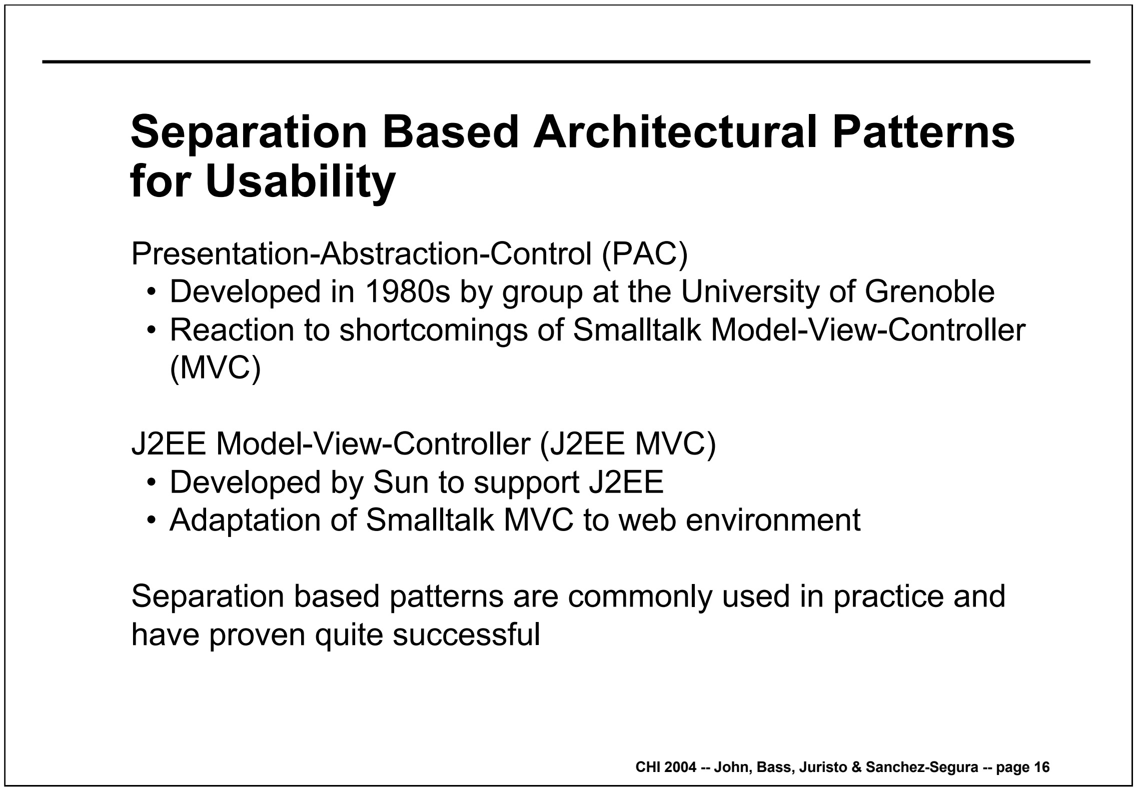
The HCI techniques supporting detailed design of the user interface are all based on iterative design

- i.e., design, test (analyze or measure), change, and re-test.

Once software has been designed, iteration implies change.

Software engineers plan for change through isolating the section to be changed (separation).

In detailed design, the items to be separated are those relating to presentation, input, possibly dialog.



Separation Based Architectural Patterns for Usability

Presentation-Abstraction-Control (PAC)
- Developed in 1980s by group at the University of Grenoble
- Reaction to shortcomings of Smalltalk Model-View-Controller (MVC)

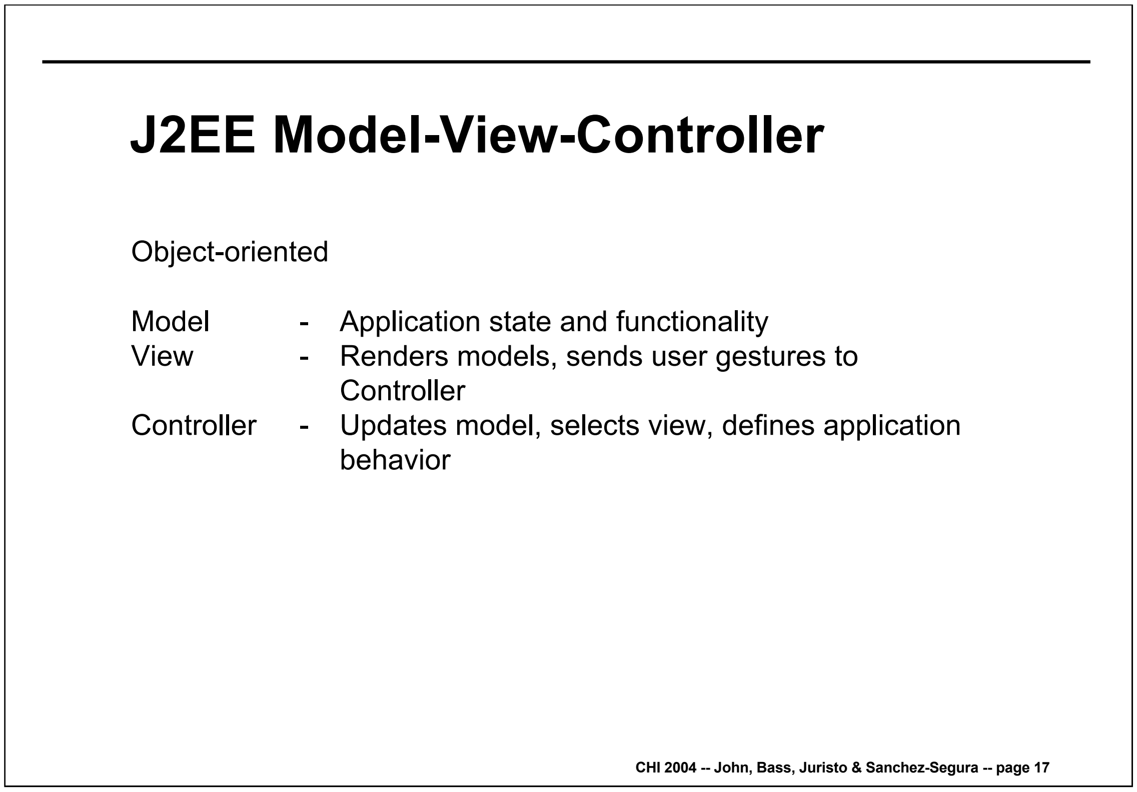
J2EE Model-View-Controller (J2EE MVC)
- Developed by Sun to support J2EE
- Adaptation of Smalltalk MVC to web environment

Separation based patterns are commonly used in practice and have proven quite successful

PAC is documented in:

Buschmann, F., Meuneir, R, Rohnert, H., Sommerlad, P. and Stal, M., (1996) *Pattern-Oriented Software Architecture, A System of Patterns*, Chichester, Eng: John Wiley and Sons.

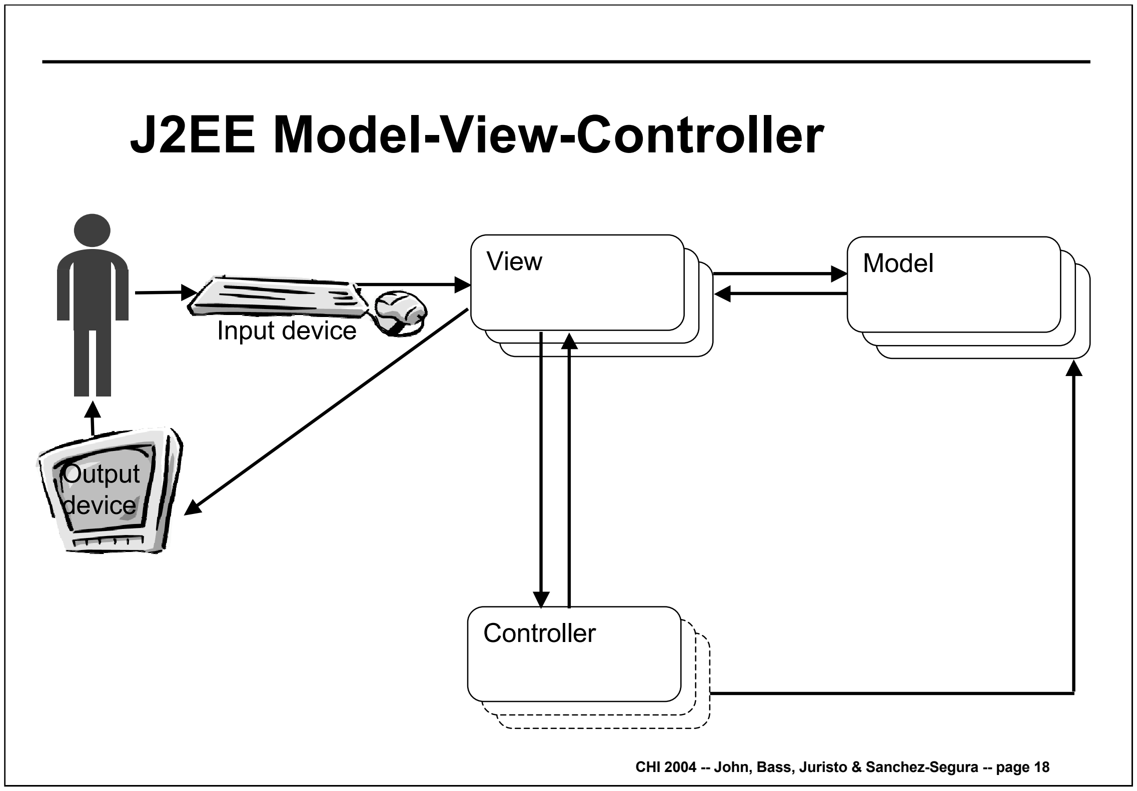
J2EE-MVC is documented at
http://java.sun.com/blueprints/patterns/MVC-detailed.html



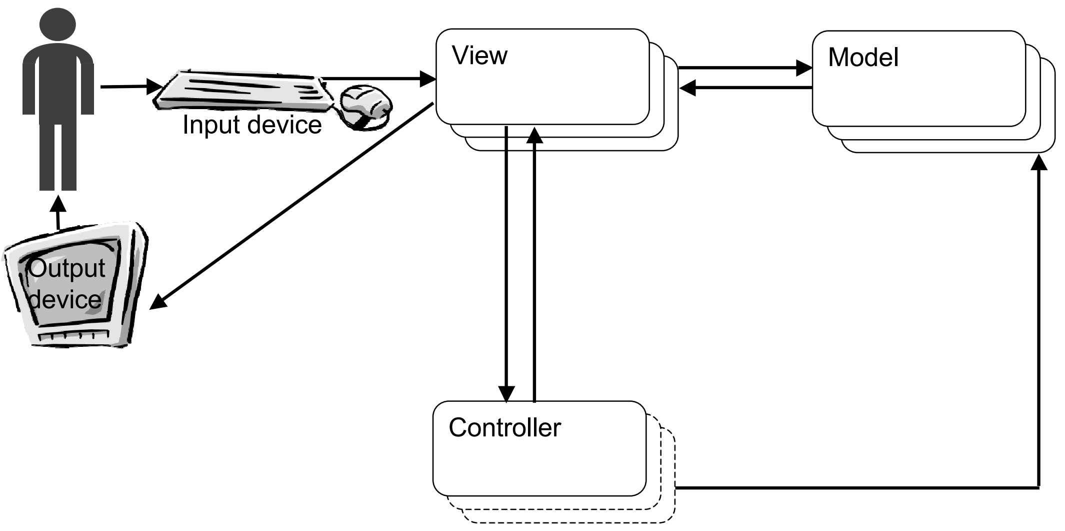
J2EE Model-View-Controller

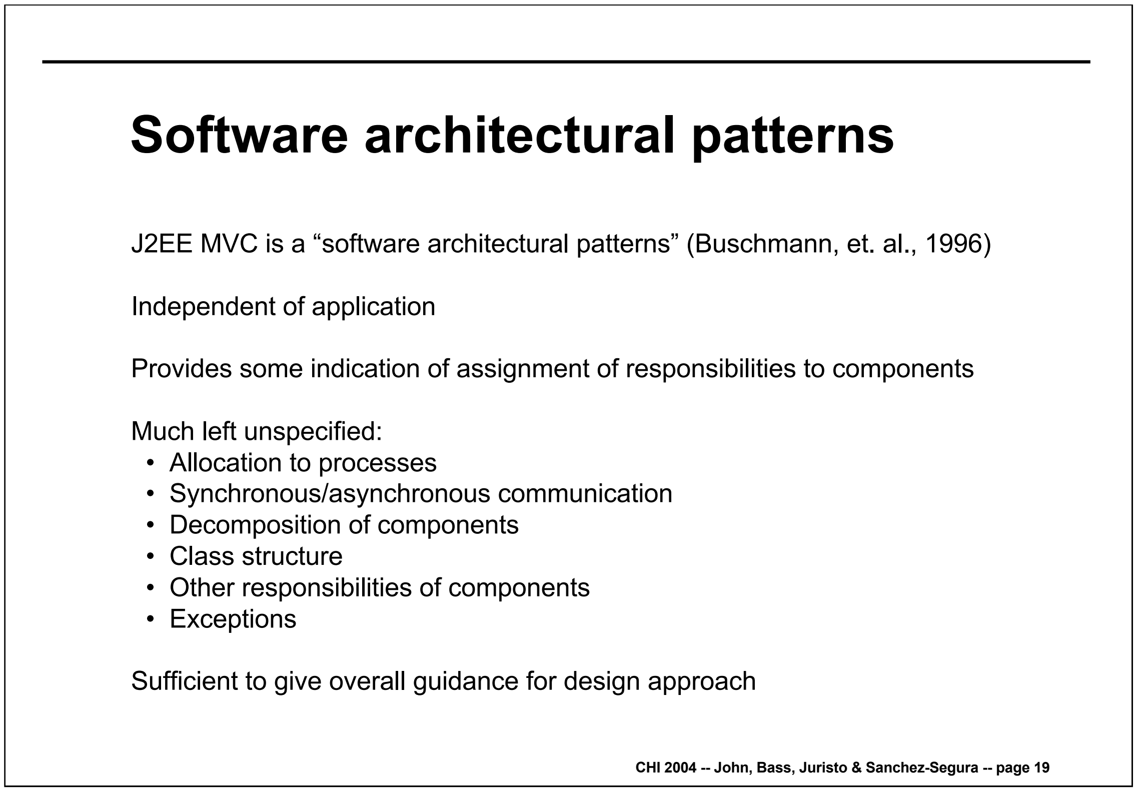
Object-oriented

Model	-	Application state and functionality
View	-	Renders models, sends user gestures to Controller
Controller	-	Updates model, selects view, defines application behavior



J2EE Model-View-Controller





Software architectural patterns

J2EE MVC is a “software architectural patterns” (Buschmann, et. al., 1996)

Independent of application

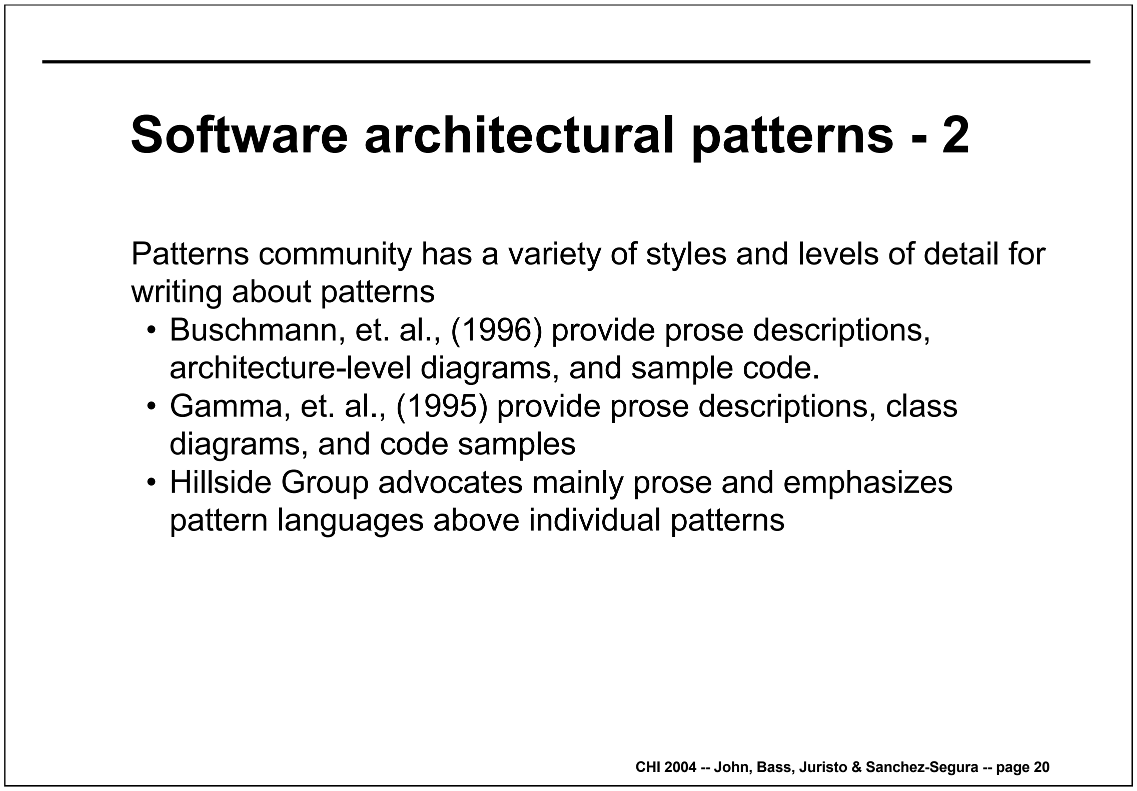
Provides some indication of assignment of responsibilities to components

Much left unspecified:
- Allocation to processes
- Synchronous/asynchronous communication
- Decomposition of components
- Class structure
- Other responsibilities of components
- Exceptions

Sufficient to give overall guidance for design approach

CHI 2004 -- John, Bass, Juristo & Sanchez-Segura -- page 19

Buschmann, F., Meuneir, R, Rohnert, H., Sommerlad, P. and Stal, M., (1996) *Pattern-Oriented Software Architecture, A System of Patterns*, Chichester, Eng: John Wiley and Sons.



Software architectural patterns - 2

Patterns community has a variety of styles and levels of detail for writing about patterns

- Buschmann, et. al., (1996) provide prose descriptions, architecture-level diagrams, and sample code.
- Gamma, et. al., (1995) provide prose descriptions, class diagrams, and code samples
- Hillside Group advocates mainly prose and emphasizes pattern languages above individual patterns

CHI 2004 -- John, Bass, Juristo & Sanchez-Segura -- page 20

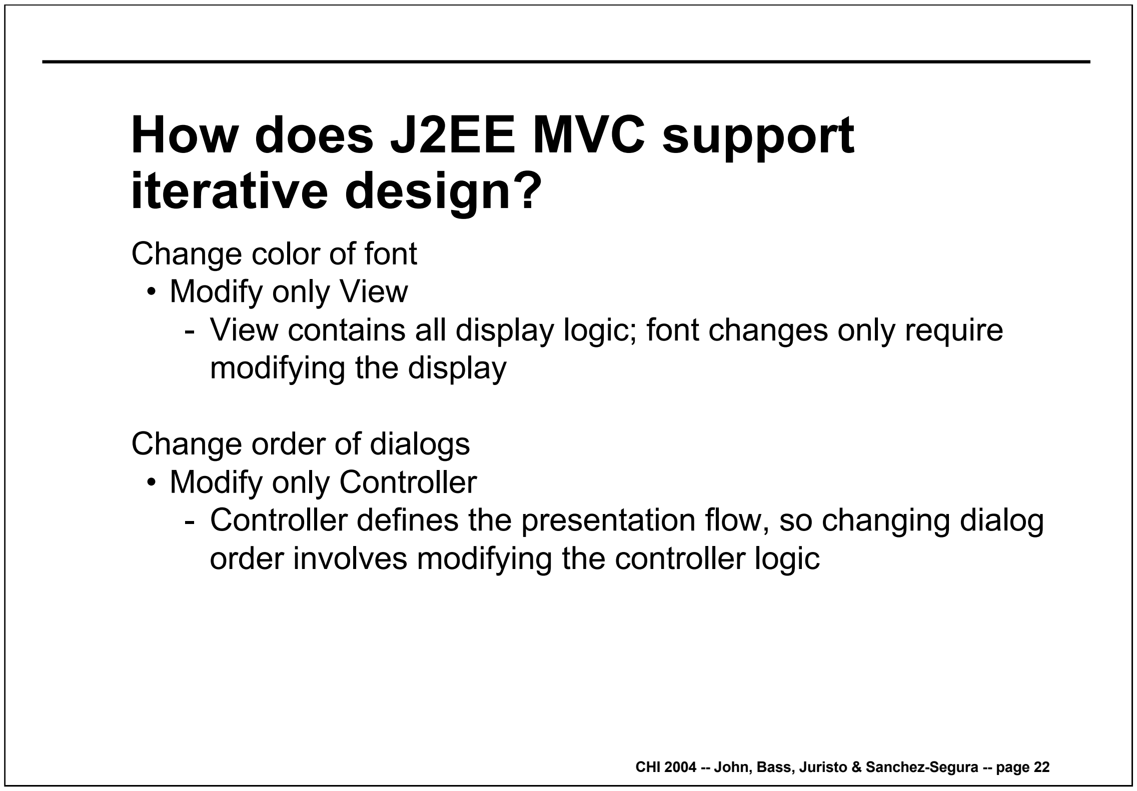
Buschmann, F., Meuneir, R, Rohnert, H., Sommerlad, P. and Stal, M., (1996) *Pattern-Oriented Software Architecture, A System of Patterns*, Chichester, Eng: John Wiley and Sons.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995). *Design Patterns.* Boston, Massachusetts: Addison-Wesley.

Information about the Hillside Group and patterns and pattern languages can be found at http://www.hillside.net/

Why separation-based architectural patterns are not sufficient for interactive systems

Remember iterative design?



How does J2EE MVC support iterative design?

Change color of font
- Modify only View
  - View contains all display logic; font changes only require modifying the display

Change order of dialogs
- Modify only Controller
  - Controller defines the presentation flow, so changing dialog order involves modifying the controller logic

CHI 2004 -- John, Bass, Juristo & Sanchez-Segura -- page 22

What happens to other usability changes?

Add the ability to cancel a long-running command

- Requires modification of all three modules
  - View – must have cancel button or other means for user to specify cancel
  - Controller – logic to respond to the View’s menu selection and execute the appropriate Model function
  - Model – free allocated resources, etc.

Shortcomings of separation patterns for solving the "We can't change THAT!" problem

With respect to adding the ability to cancel

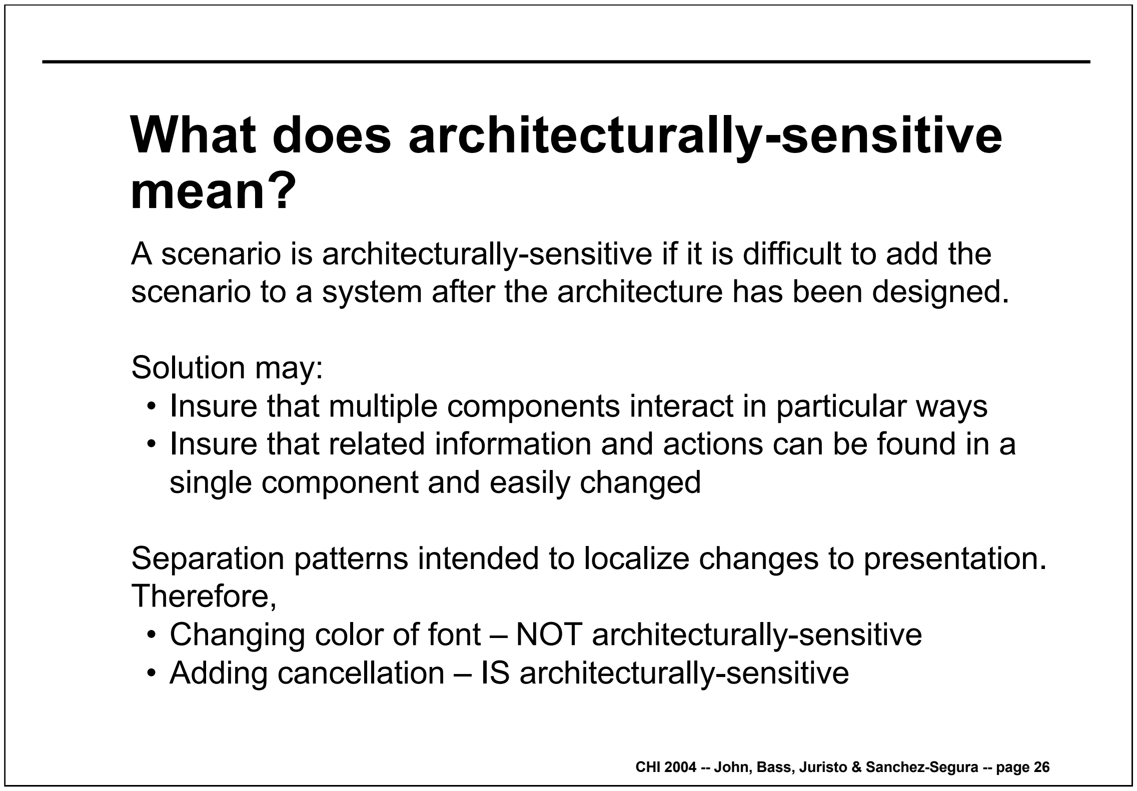
- Involved all components
- Not much localization
- If requirement for cancel discovered late, then will require extensive modification to the architecture.

Beyond separation-based architectural patterns: The Usability-Supporting Architectural Patterns (USAP) Approach

Our goal is to provide software designers and usability specialist tools to recognize and prevent common usability problems that are not supported by separation.

We are doing this by:

- Identifying those aspects of usability that are “architecturally sensitive” and embodying them in small scenarios
- Providing a way to reason about the forces acting on architecture design in these scenarios
- Providing checklist of important software responsibilities and possible architecture patterns to satisfy these scenarios



What does architecturally-sensitive mean?

A scenario is architecturally-sensitive if it is difficult to add the scenario to a system after the architecture has been designed.

Solution may:

- Insure that multiple components interact in particular ways
- Insure that related information and actions can be found in a single component and easily changed

Separation patterns intended to localize changes to presentation. Therefore,

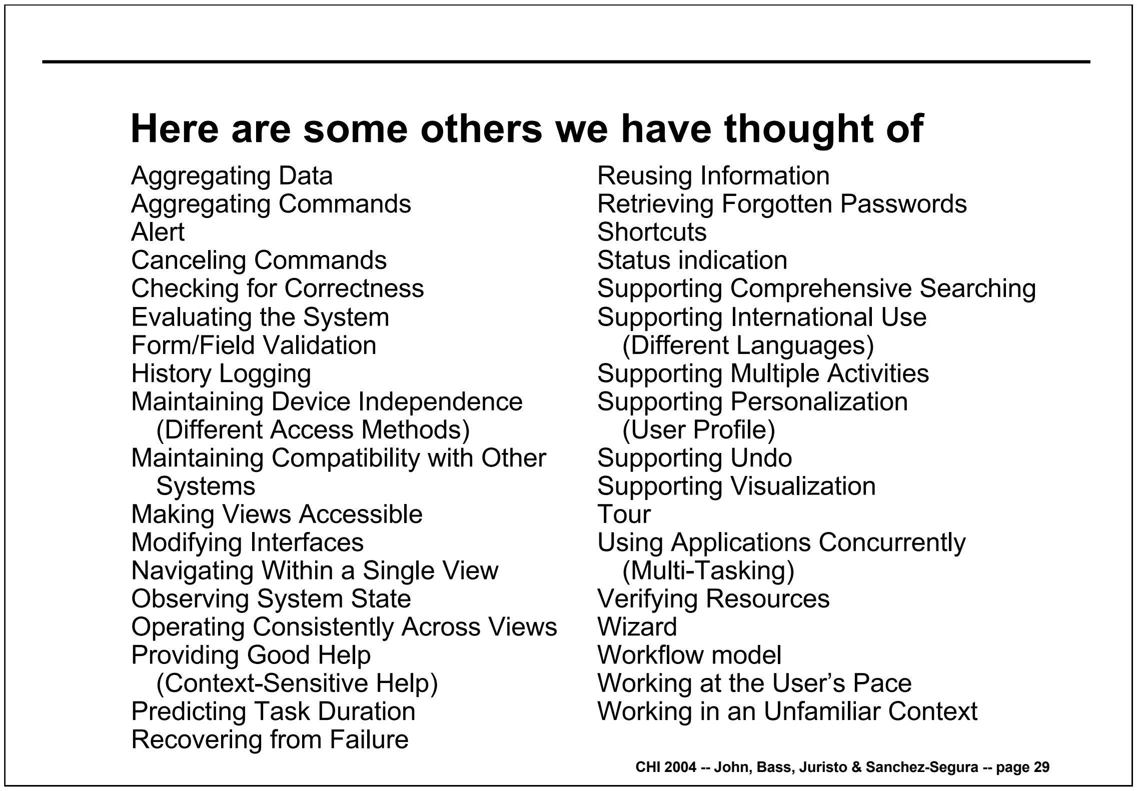
- Changing color of font – NOT architecturally-sensitive
- Adding cancellation – IS architecturally-sensitive

CHI 2004 -- John, Bass, Juristo & Sanchez-Segura -- page 26

An architecturally-sensitive scenario: Canceling commands

The user issues a command then changes his or her mind, wanting to stop the operation and return the software to its pre-operation state. It doesn't matter why the user wants to stop; he or she could have made a mistake, the system could be unresponsive, or the environment could have changed.

What other architecturally-sensitive scenarios can you think of?



Here are some others we have thought of

Aggregating Data
Aggregating Commands
Alert
Canceling Commands
Checking for Correctness
Evaluating the System
Form/Field Validation
History Logging
Maintaining Device Independence
(Different Access Methods)
Maintaining Compatibility with Other Systems
Making Views Accessible
Modifying Interfaces
Navigating Within a Single View
Observing System State
Operating Consistently Across Views
Providing Good Help
(Context-Sensitive Help)
Predicting Task Duration
Recovering from Failure

Reusing Information
Retrieving Forgotten Passwords
Shortcuts
Status indication
Supporting Comprehensive Searching
Supporting International Use
(Different Languages)
Supporting Multiple Activities
Supporting Personalization
(User Profile)
Supporting Undo
Supporting Visualization
Tour
Using Applications Concurrently
(Multi-Tasking)
Verifying Resources
Wizard
Workflow model
Working at the User's Pace
Working in an Unfamiliar Context

CHI 2004 -- John, Bass, Juristo & Sanchez-Segura -- page 29

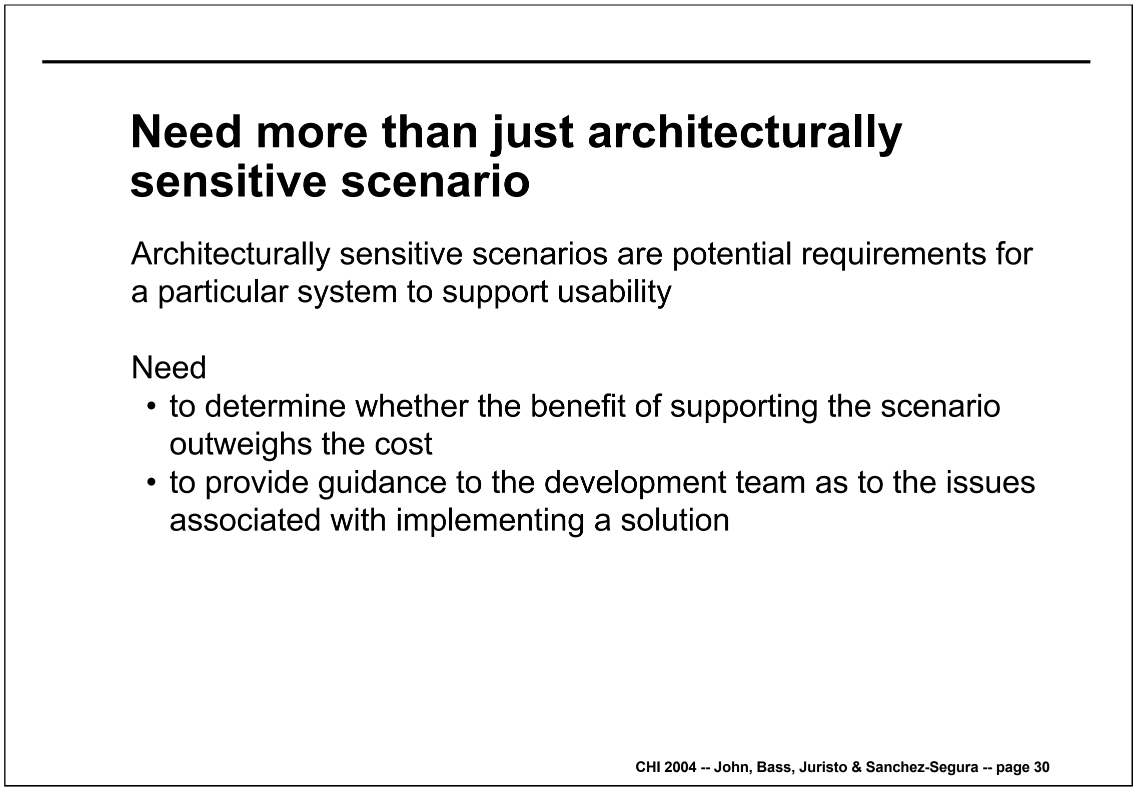
This list of architecturally-sensitive usability scnearios is compiled from

Bass, L., John, B. E., & Kates, J. (2001). *Achieving usability through software architecture* (CMU/SEI-2001-TR-005). Pittsburgh, PA: Software Engineering Institute.
`http://www.sei.cmu.edu/publications/documents/01.reports/01tr005.html`

and

Juristo , N., Moreno, A. M., & Sanchez, M. (2003) Deliverable D.3.4. Techniques, patterns and styles for architecture-level usability improvement. - ESPRIT project (IST-2001-32298)
`http://www.ls.fi.upm.es/status/results/deliverables.html`

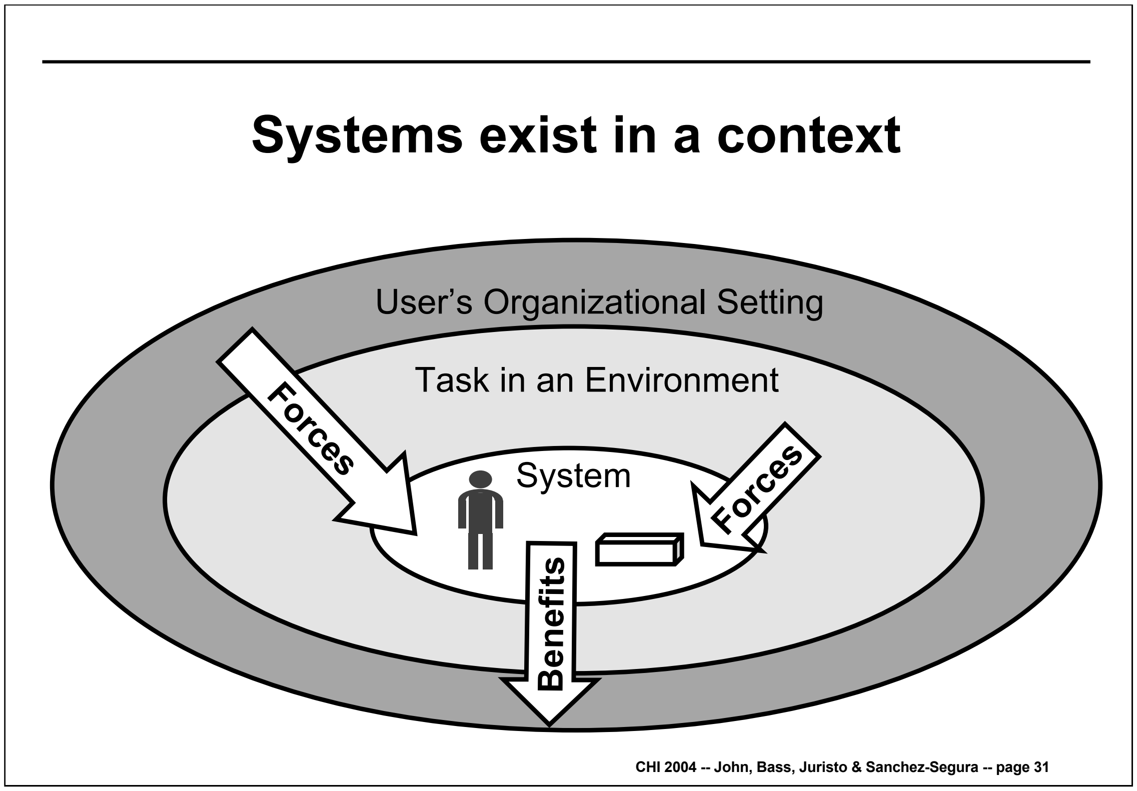
An excerpt of Bass, John & Kates (2001), describing a set of architecturally-sensitive suability scenarios can be found in Appendix I.



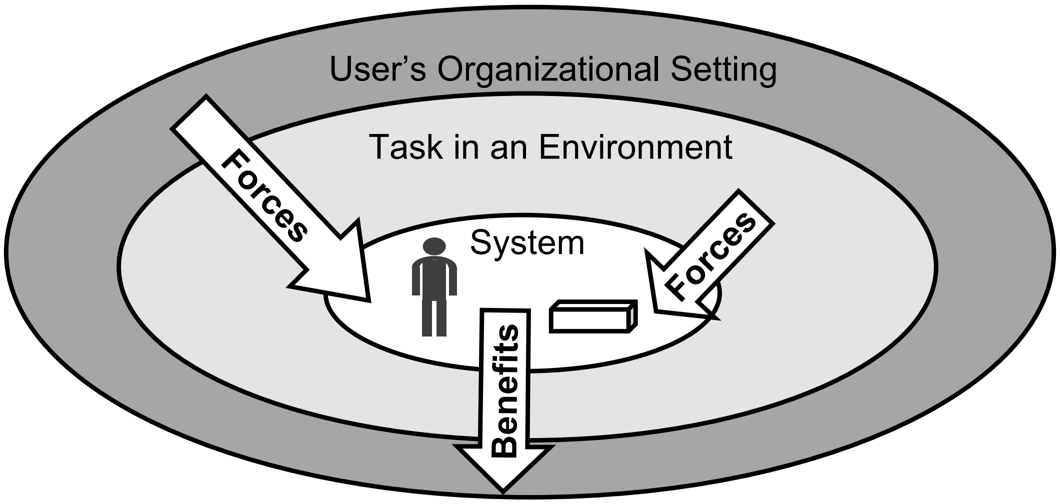
Need more than just architecturally sensitive scenario

Architecturally sensitive scenarios are potential requirements for a particular system to support usability

Need
- to determine whether the benefit of supporting the scenario outweighs the cost
- to provide guidance to the development team as to the issues associated with implementing a solution



Systems exist in a context



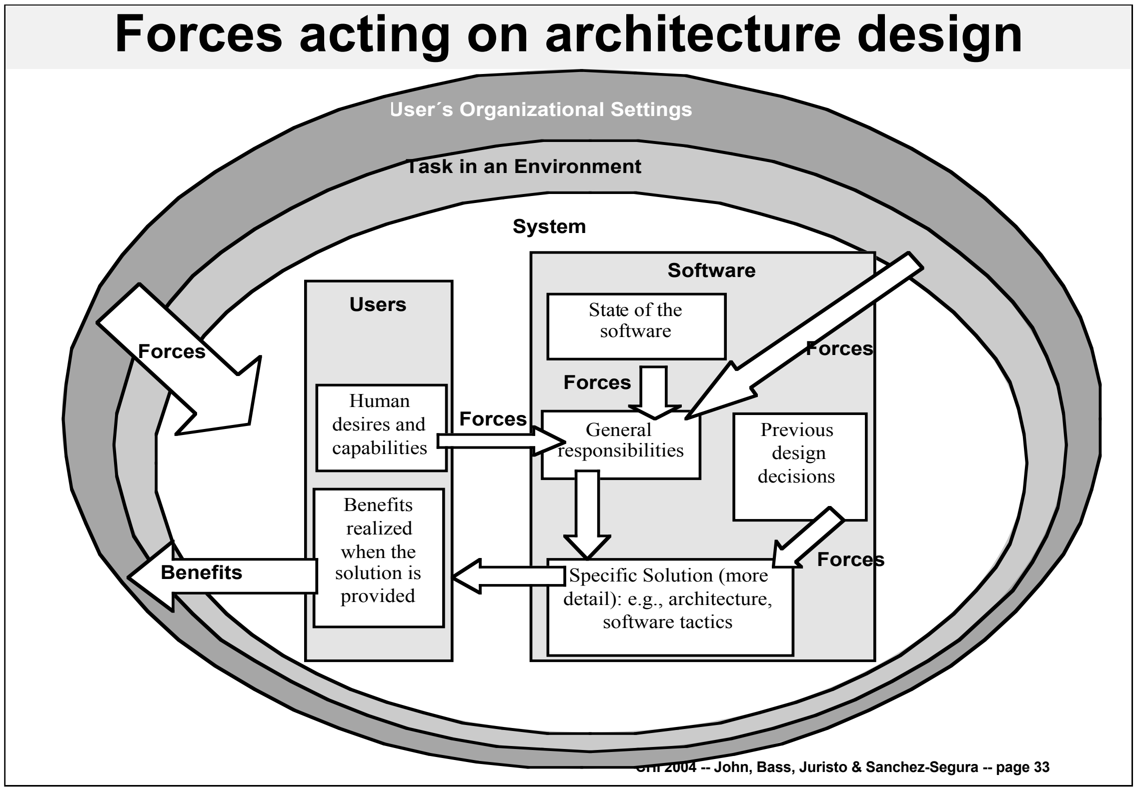
CHI 2004 -- John, Bass, Juristo & Sanchez-Segura -- page 31

Context for computer system

Computer systems fulfill “business” goals
- “Business goals” could be mission, academic, entertainment, etc.
- User using the system creates certain benefits for the “organization” that created it
- Creating system has costs.

Cost/Benefit
- Implementation support for total scenario
- Implementation support for pieces of the scenario

But more detail is necessary to be able to understand cost/benefit and implications of implementation



Forces acting on architecture design

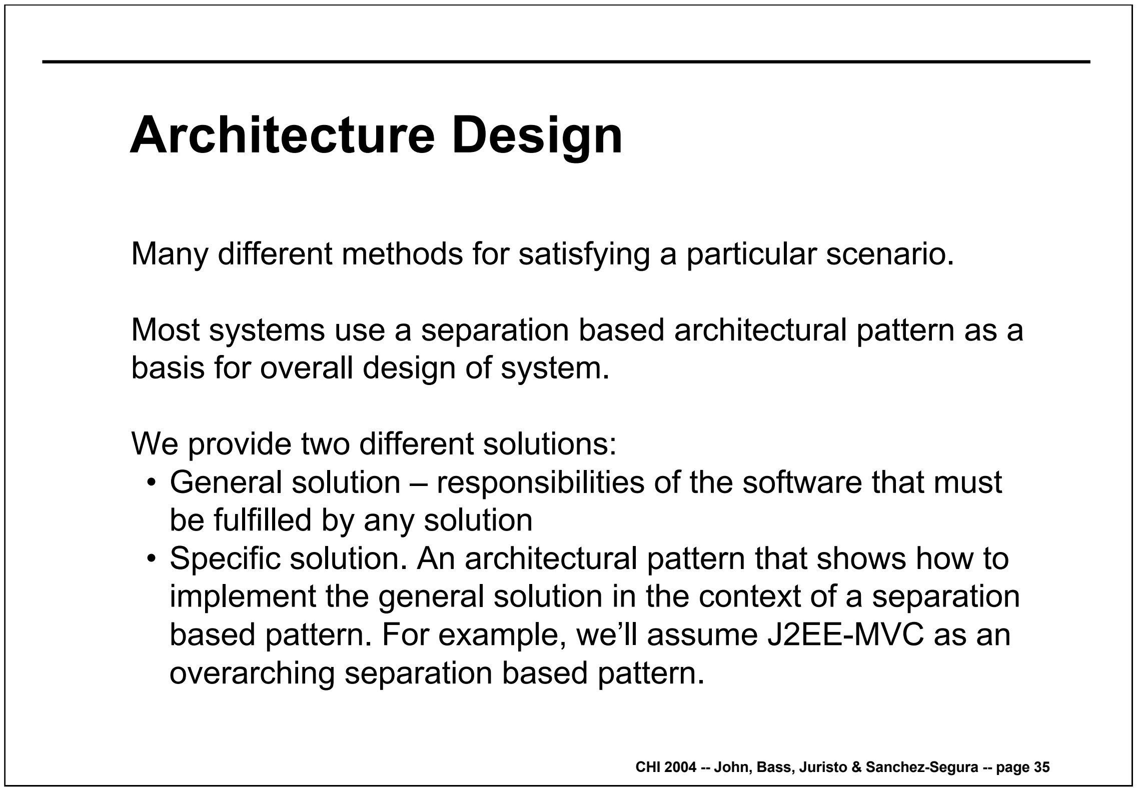
User´s Organizational Settings
Task in an Environment
System
Users
Software
State of the software
Forces
Human desires and capabilities
Forces
General responsibilities
Previous design decisions
Forces
Forces
Benefits
Benefits realized when the solution is provided
Specific Solution (more detail): e.g., architecture, software tactics
Forces
CHI 2004 -- John, Bass, Juristo & Sanchez-Segura -- page 33

Reasoning about architecture design

Differing forces motivate particular aspects of solution.

Forces come from three sources:

- Task and environment in which user is operating.
  - E.g., Cancel is only useful if operation is long running.
- Human desires and capabilities.
  - E.g., User makes mistakes, Cancel allows one type of correction of mistake.
- State of the software.
  - E.g., Networks fail. Giving the user the ability to cancel may prevent the user from being blocked because of this failure.



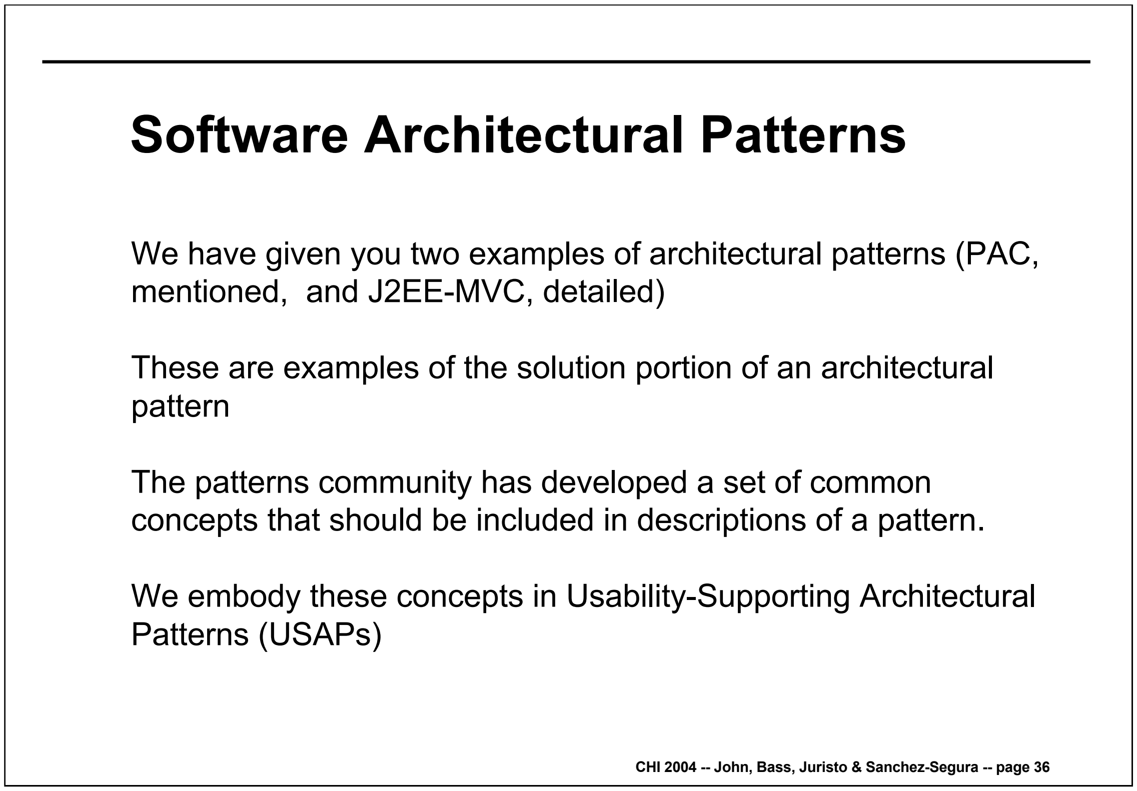
Architecture Design

Many different methods for satisfying a particular scenario.

Most systems use a separation based architectural pattern as a basis for overall design of system.

We provide two different solutions:
- General solution – responsibilities of the software that must be fulfilled by any solution
- Specific solution. An architectural pattern that shows how to implement the general solution in the context of a separation based pattern. For example, we'll assume J2EE-MVC as an overarching separation based pattern.

CHI 2004 -- John, Bass, Juristo & Sanchez-Segura -- page 35



Software Architectural Patterns

We have given you two examples of architectural patterns (PAC, mentioned, and J2EE-MVC, detailed)

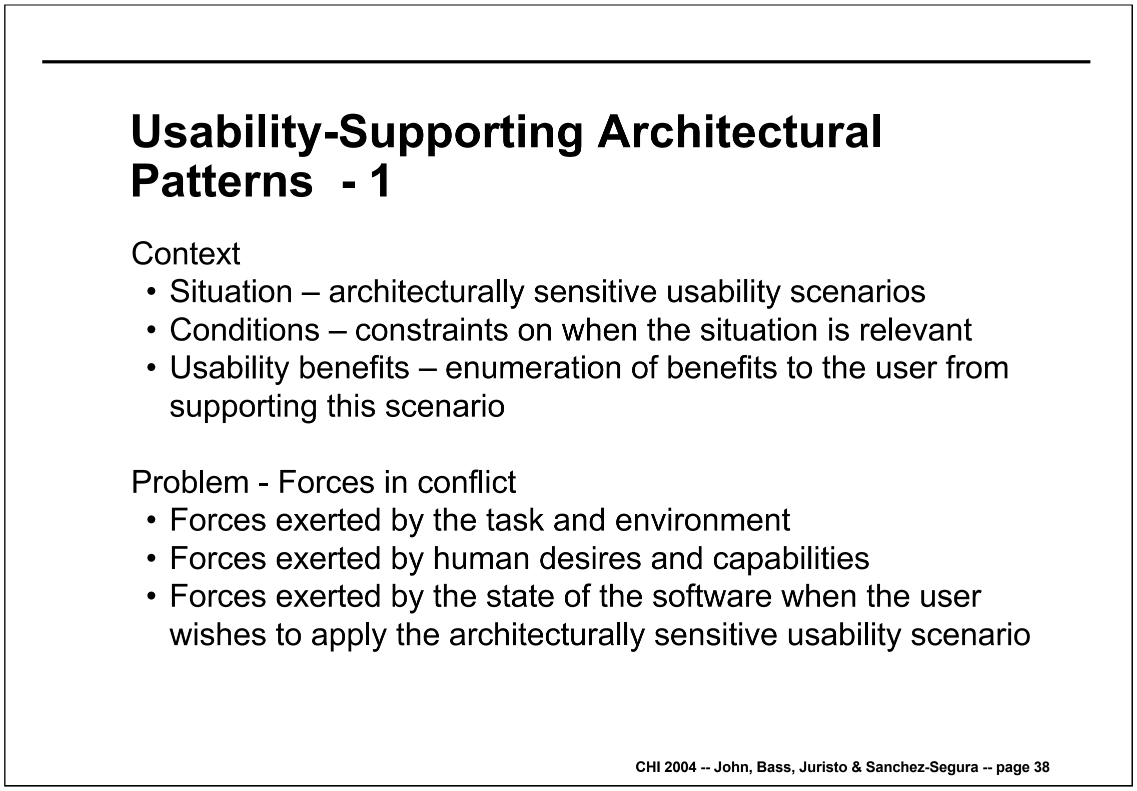
These are examples of the solution portion of an architectural pattern

The patterns community has developed a set of common concepts that should be included in descriptions of a pattern.

We embody these concepts in Usability-Supporting Architectural Patterns (USAPs)

CHI 2004 -- John, Bass, Juristo & Sanchez-Segura -- page 36

Break



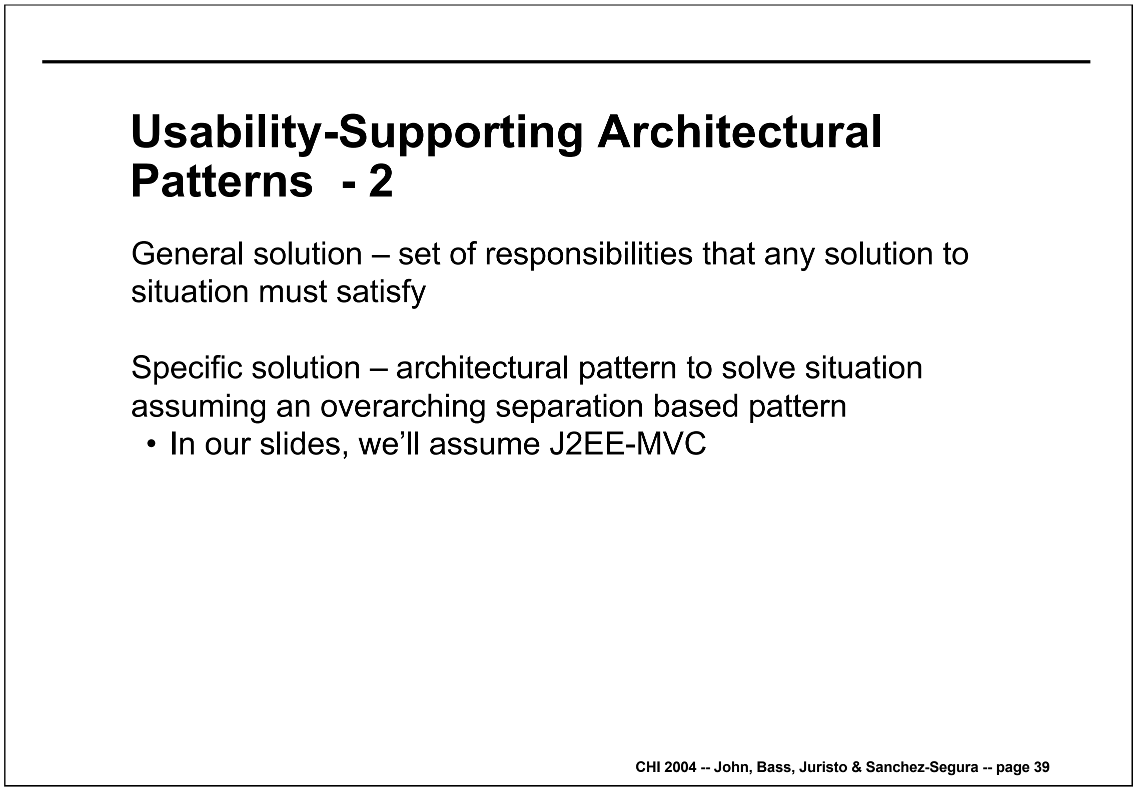
Usability-Supporting Architectural Patterns - 1

Context

- Situation – architecturally sensitive usability scenarios
- Conditions – constraints on when the situation is relevant
- Usability benefits – enumeration of benefits to the user from supporting this scenario

Problem - Forces in conflict

- Forces exerted by the task and environment
- Forces exerted by human desires and capabilities
- Forces exerted by the state of the software when the user wishes to apply the architecturally sensitive usability scenario



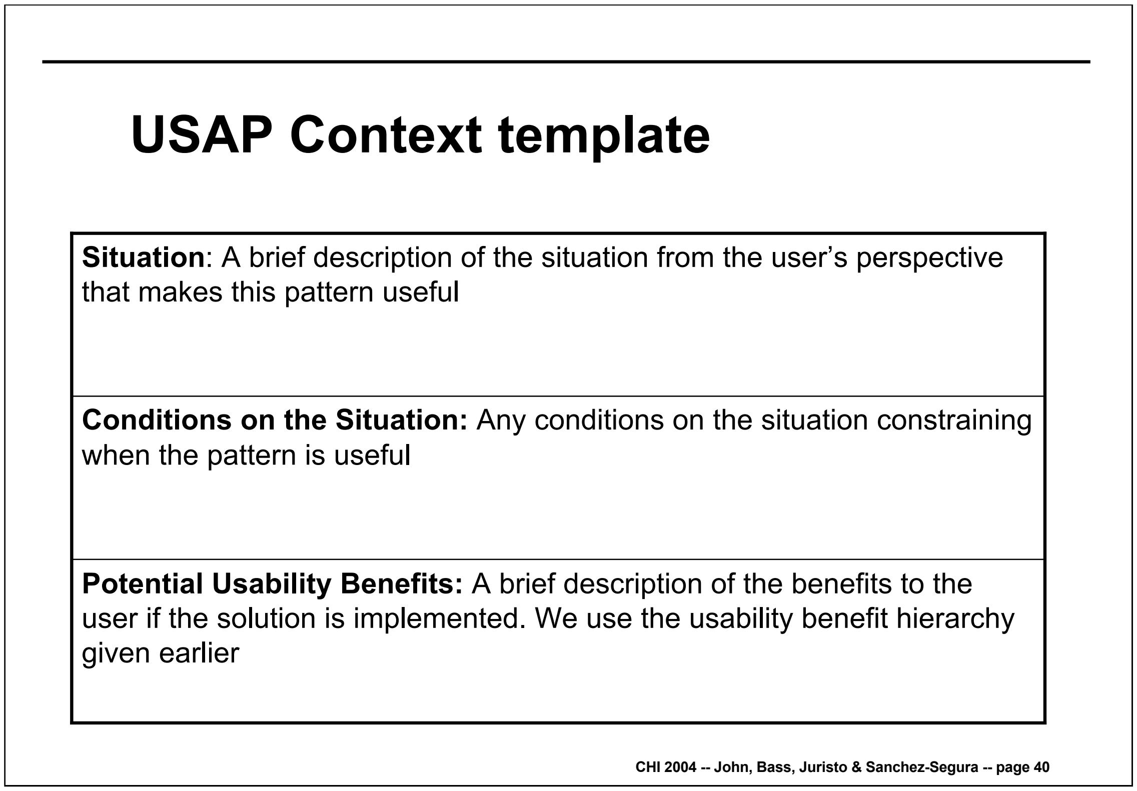
Usability-Supporting Architectural Patterns - 2

General solution – set of responsibilities that any solution to situation must satisfy

Specific solution – architectural pattern to solve situation assuming an overarching separation based pattern

- In our slides, we'll assume J2EE-MVC

CHI 2004 -- John, Bass, Juristo & Sanchez-Segura -- page 39



USAP Context template

Situation: A brief description of the situation from the user's perspective that makes this pattern useful
Conditions on the Situation: Any conditions on the situation constraining when the pattern is useful
Potential Usability Benefits: A brief description of the benefits to the user if the solution is implemented. We use the usability benefit hierarchy given earlier

CHI 2004 -- John, Bass, Juristo & Sanchez-Segura -- page 40

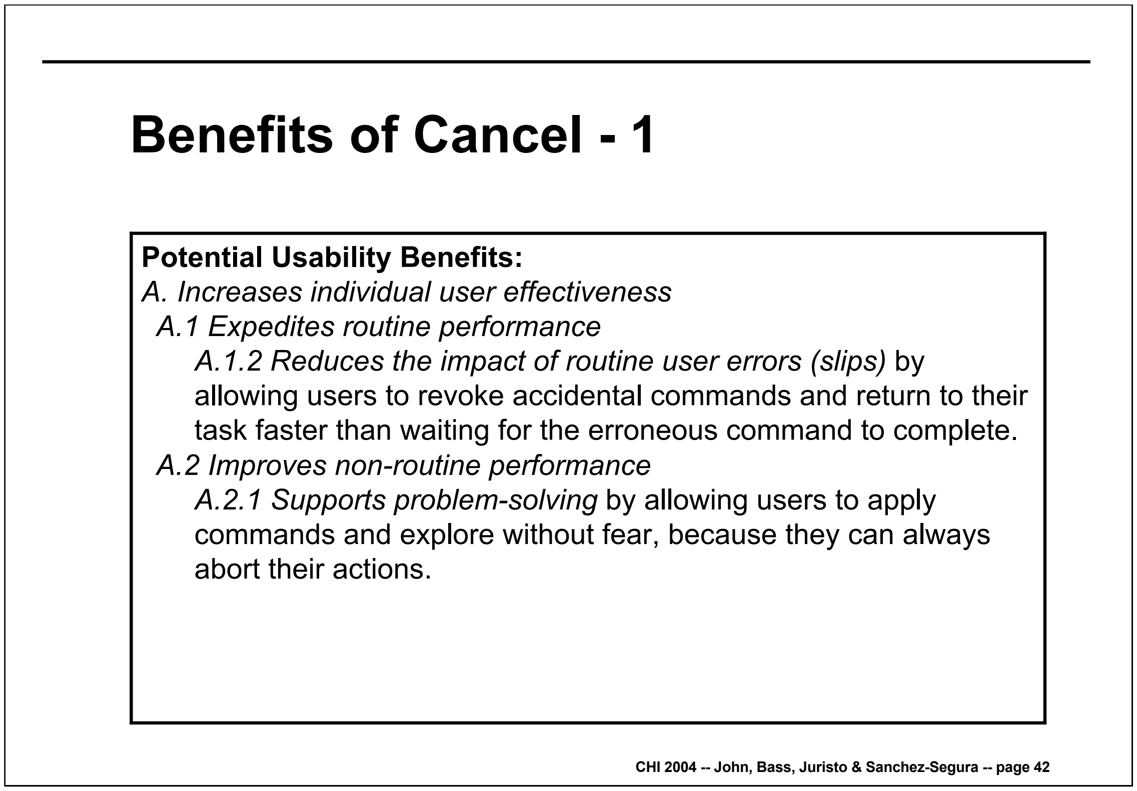
An excerpt of Bass, John & Kates (2001), describing a usability benefit hierarchy can be found in Appendix II.

The full USAP template can be found in Appendix III.

USAP Context for Cancel - 1

Situation: The user issues a command then changes his or her mind, wanting to stop the operation and return the software to its pre-operation state. It doesn’t matter why the user wants to stop; he or she could have made a mistake, the system could be unresponsive, or the environment could have changed.

Conditions on the Situation: A user is working in a system where the software has long-running commands, i.e., more than one second.
The cancellation command could be explicitly issued by the user, or through some sensing of the environment (e.g., a child’s hand in a power car window).



Benefits of Cancel - 1

Potential Usability Benefits:

A. Increases individual user effectiveness

- *A.1 Expedites routine performance*
  - *A.1.2 Reduces the impact of routine user errors (slips)* by allowing users to revoke accidental commands and return to their task faster than waiting for the erroneous command to complete.
- *A.2 Improves non-routine performance*
  - *A.2.1 Supports problem-solving* by allowing users to apply commands and explore without fear, because they can always abort their actions.

CHI 2004 -- John, Bass, Juristo & Sanchez-Segura -- page 42

Benefits of Cancel – 2

Potential Usability Benefits:

A. Increases individual user effectiveness

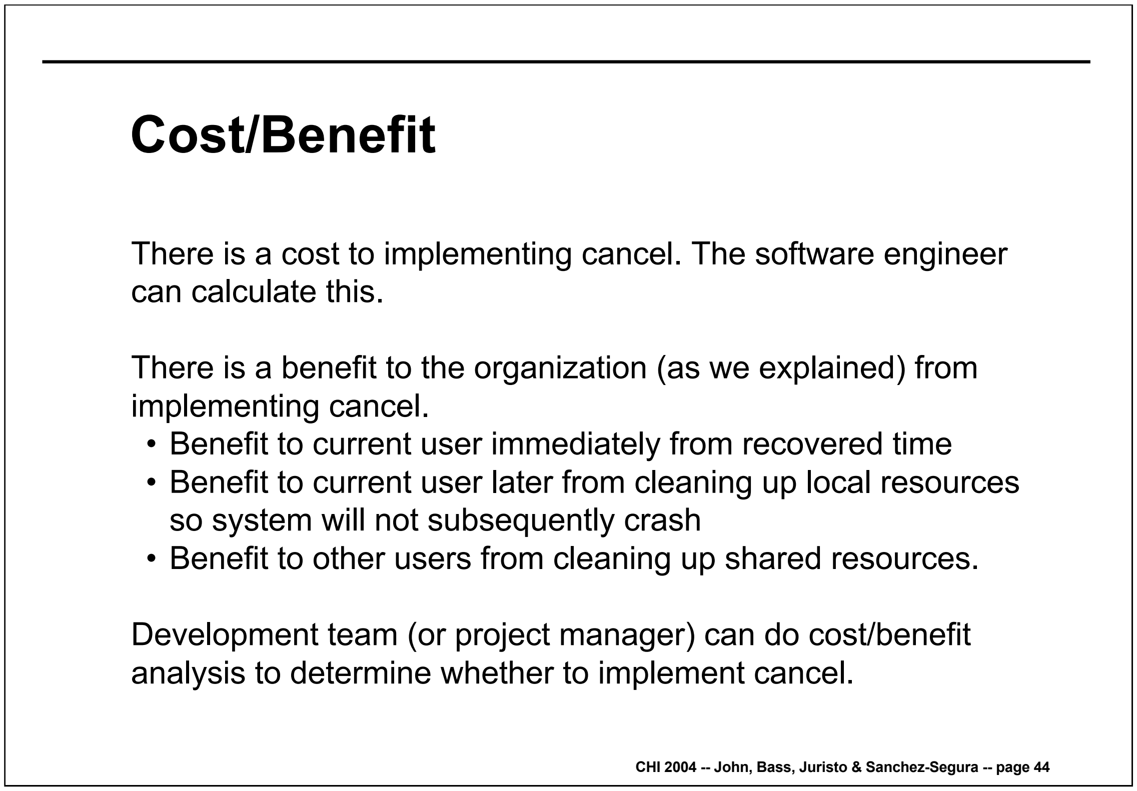
A.3 Reduces the impact of user errors caused by lack of knowledge (mistakes)

A.3.2 Accommodates mistakes by allowing users to abort commands they invoke through lack of knowledge and return to their task faster than waiting for the erroneous command to complete.

B. Reduces the impact of system errors

B.2 Tolerates system errors by allowing users to abort commands that aren't working properly (for example, a user cancels a download because the network is jammed).

C.Increases user confidence and comfort by allowing users to perform without fear because they can always abort their actions.



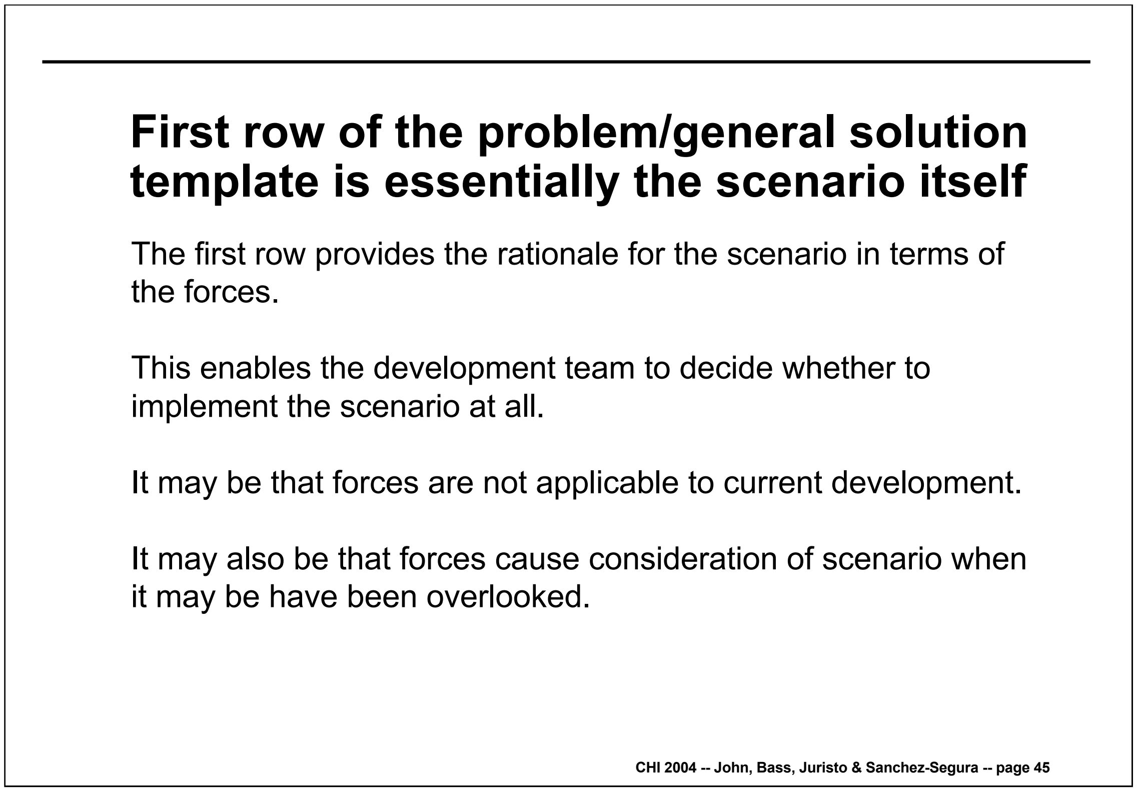
Cost/Benefit

There is a cost to implementing cancel. The software engineer can calculate this.

There is a benefit to the organization (as we explained) from implementing cancel.

- Benefit to current user immediately from recovered time
- Benefit to current user later from cleaning up local resources so system will not subsequently crash
- Benefit to other users from cleaning up shared resources.

Development team (or project manager) can do cost/benefit analysis to determine whether to implement cancel.



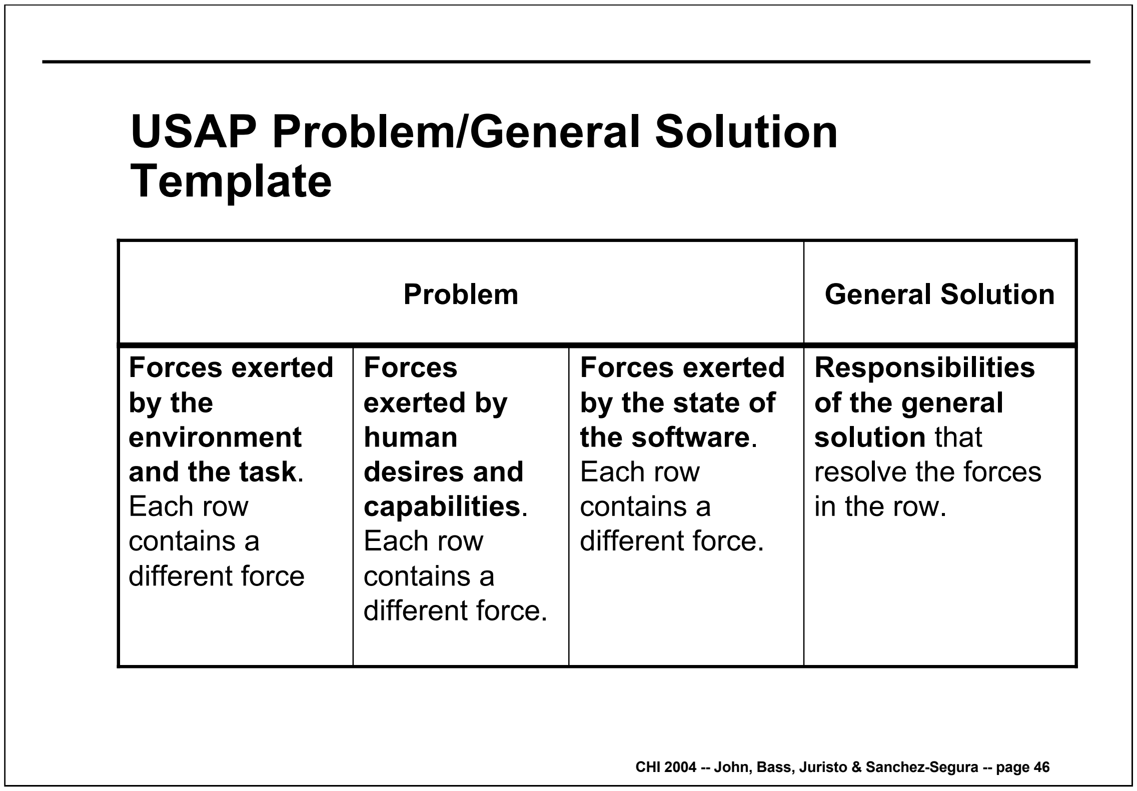
First row of the problem/general solution template is essentially the scenario itself

The first row provides the rationale for the scenario in terms of the forces.

This enables the development team to decide whether to implement the scenario at all.

It may be that forces are not applicable to current development.

It may also be that forces cause consideration of scenario when it may be have been overlooked.



USAP Problem/General Solution Template

Problem			General Solution
Forces exerted by the environment and the task. Each row contains a different force	**Forces exerted by human desires and capabilities**. Each row contains a different force.	**Forces exerted by the state of the software**. Each row contains a different force.	**Responsibilities of the general solution** that resolve the forces in the row.

CHI 2004 -- John, Bass, Juristo & Sanchez-Segura -- page 46

Cancel Problem/General Solution: Responsibility R1 is essentially the scenario itself

Problem			General Solution
Networks are sometimes unresponsive. Sometimes changes in the environment require the system to terminate.	Users slip or make mistakes, or explore commands and then change their minds, but do not want to wait for the command to complete.	Software is sometimes unresponsive	R1. Must provide a means to cancel a command

Template Problem/General Solution - other rows

Each subsequent row of the problem general solution template provides rationale for one or more responsibilities.

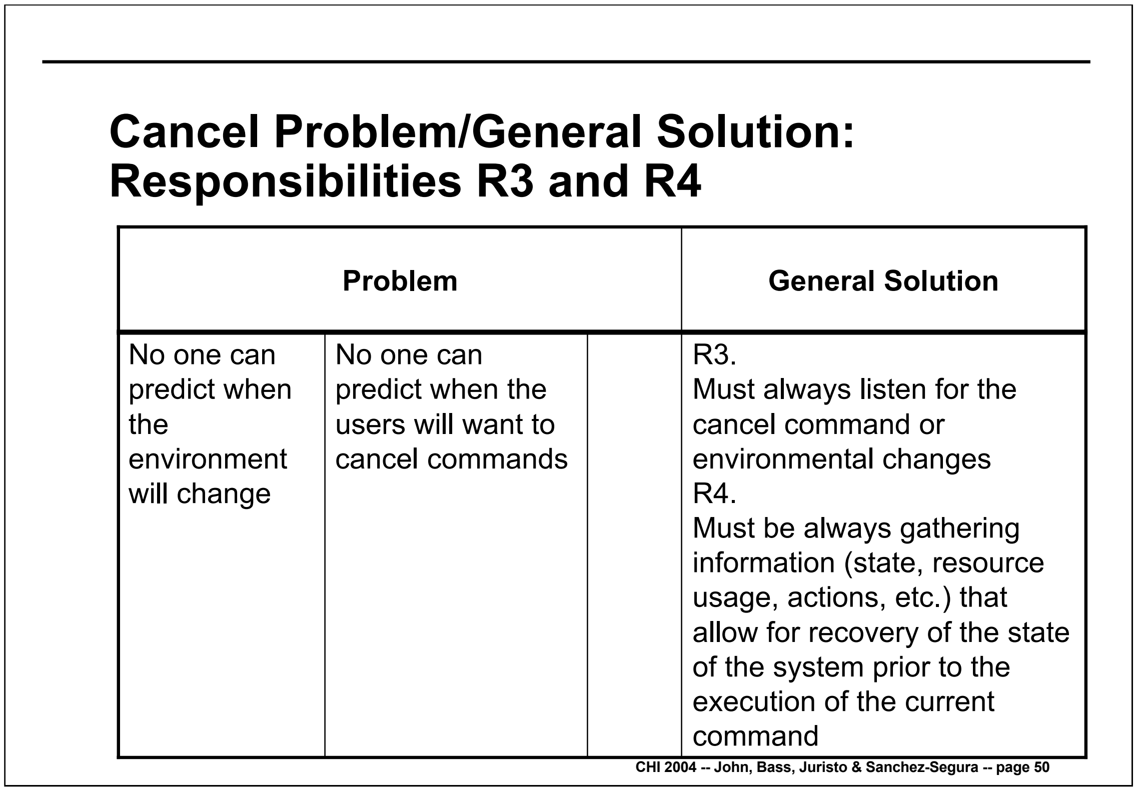
Usually one row per responsibility, but sometimes rationale for multiple responsibilities are the same and so multiple responsibilities are included in one row.

Allows development team to understand reason for responsibility and make cost/benefit decisions about:

- Necessity
- Utility

Cancel Problem/General Solution: Responsibility R2

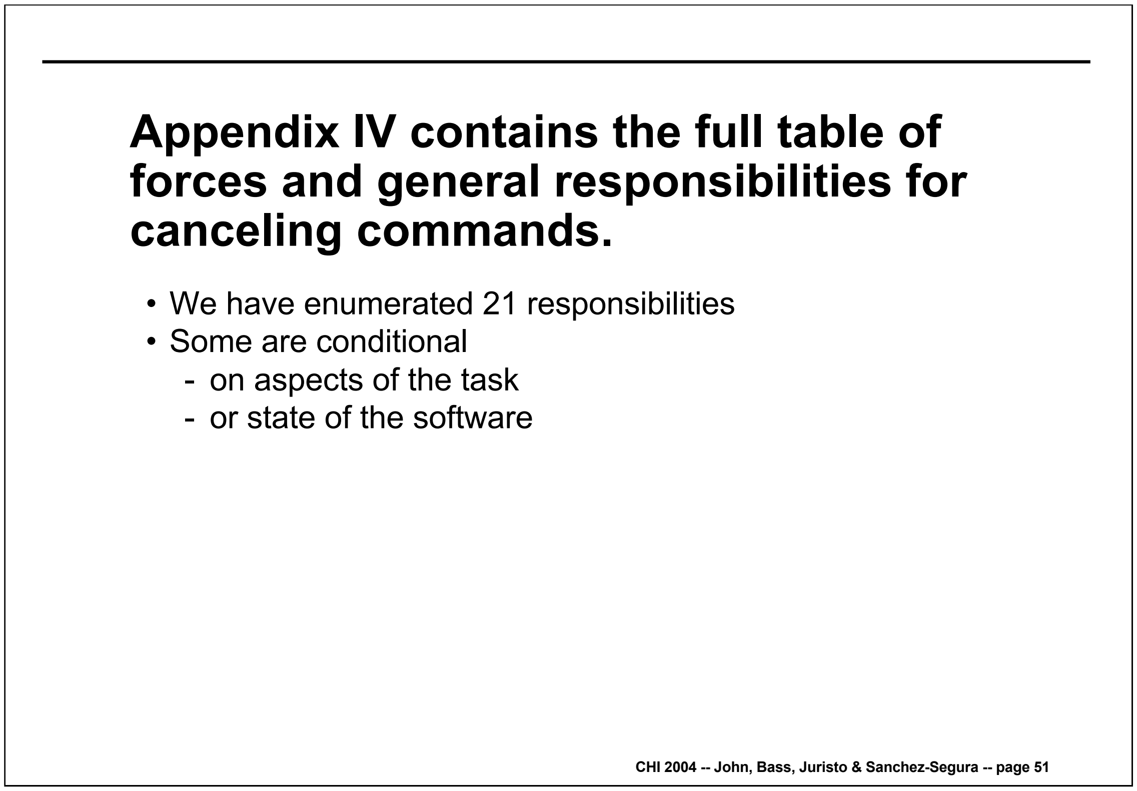
Problem			General Solution
	Users have to communicate their intentions to the software through overt acts (e.g., finger movements)	Software has to receive an action from the user to do something	R2. Provide a button, menu item, keyboard shortcut and/or other means to cancel the active command.



Cancel Problem/General Solution: Responsibilities R3 and R4

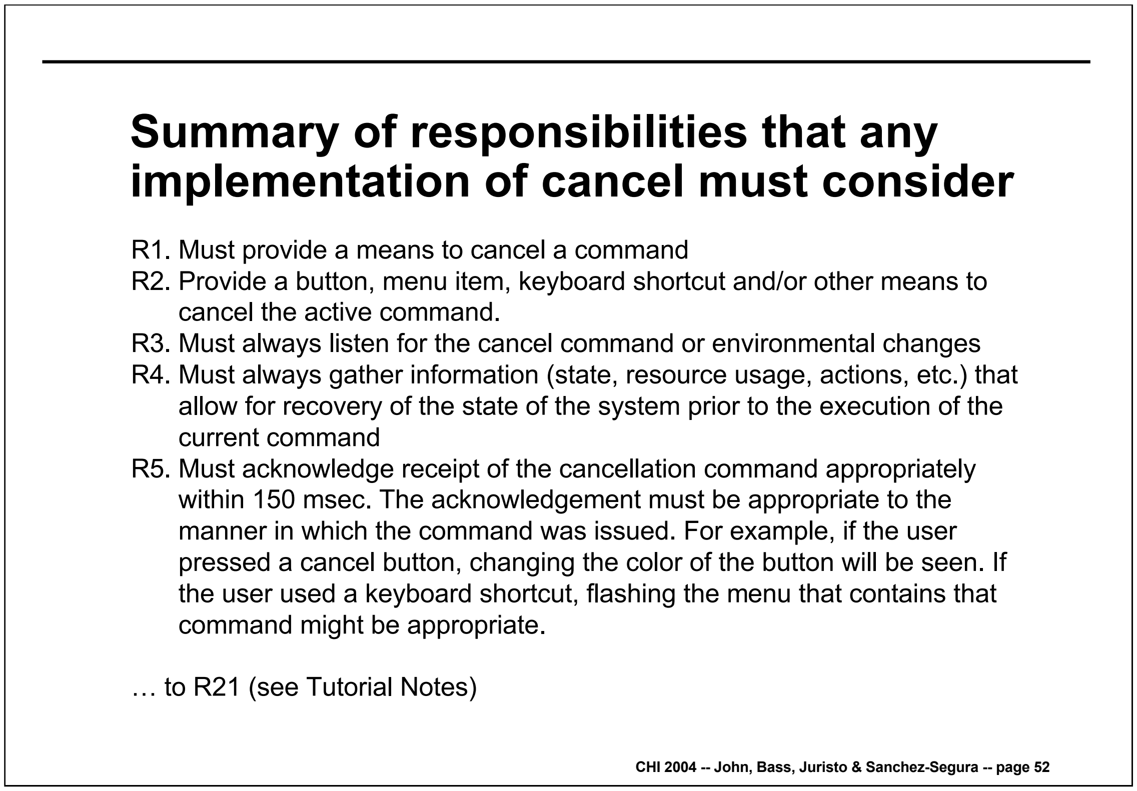
Problem			General Solution
No one can predict when the environment will change	No one can predict when the users will want to cancel commands		R3. Must always listen for the cancel command or environmental changes R4. Must be always gathering information (state, resource usage, actions, etc.) that allow for recovery of the state of the system prior to the execution of the current command

CHI 2004 -- John, Bass, Juristo & Sanchez-Segura -- page 50



Appendix IV contains the full table of forces and general responsibilities for canceling commands.

- We have enumerated 21 responsibilities
- Some are conditional
  - on aspects of the task
  - or state of the software



Summary of responsibilities that any implementation of cancel must consider

R1. Must provide a means to cancel a command
R2. Provide a button, menu item, keyboard shortcut and/or other means to cancel the active command.
R3. Must always listen for the cancel command or environmental changes
R4. Must always gather information (state, resource usage, actions, etc.) that allow for recovery of the state of the system prior to the execution of the current command
R5. Must acknowledge receipt of the cancellation command appropriately within 150 msec. The acknowledgement must be appropriate to the manner in which the command was issued. For example, if the user pressed a cancel button, changing the color of the button will be seen. If the user used a keyboard shortcut, flashing the menu that contains that command might be appropriate.

… to R21 (see Tutorial Notes)

CHI 2004 -- John, Bass, Juristo & Sanchez-Segura -- page 52

Either the command itself is responsive

R6. The command must have the ability to cancel itself (I.e., it must fulfill Responsibilities R10 to R21 (e.g., an object-oriented system would have a cancel method in each object)

Or the command itself is not responsive

R7. An active portion of the application must ask the infrastructure to cancel the command, or

R8. The infrastructure itself must provide a means to request the cancellation of the application (e.g., task manager on Windows, force quit on MacOS)

R9. If either R7 or R8, then the infrastructure must have the ability to cancel the active command (I.e., it must fulfill Responsibilities R10 to R21)

If the command has invoked collaborating processes

R10. The collaborating processes have to be informed of the cancellation of the invoking command (these processes have their own responsibilities that they must perform in response to this information, possibly treat it as a cancellation.). The information given to collaborating processes may include the request for cancellation, the progress of cancellation, and/or the completion of cancellation.

Continuation of responsibilities that any implementation of cancel must consider

Either the system is capable of rolling back all changes to the state prior to execution of the command.

R11. Restore the system back to its state prior to execution of the command.

Or the system is not capable of rolling back all changes to the state prior to execution of the command.

R12. Restore the system back to as close to the state prior to execution of the command as possible

R13. Inform the user of the difference between the prior state and the restored state.

R14. Resources that can be freed must be freed

If any resources are not capable of being freed, then,

R15. Inform the user of the partially-restored resources in a manner that they will see it.

For critical tasks with incomplete state or resource restoration,

R16. Require acknowledgement from the user that they are aware of the partially-restored nature of the cancellation.

R17. Return control to the user, or not, depending on the forces from the task

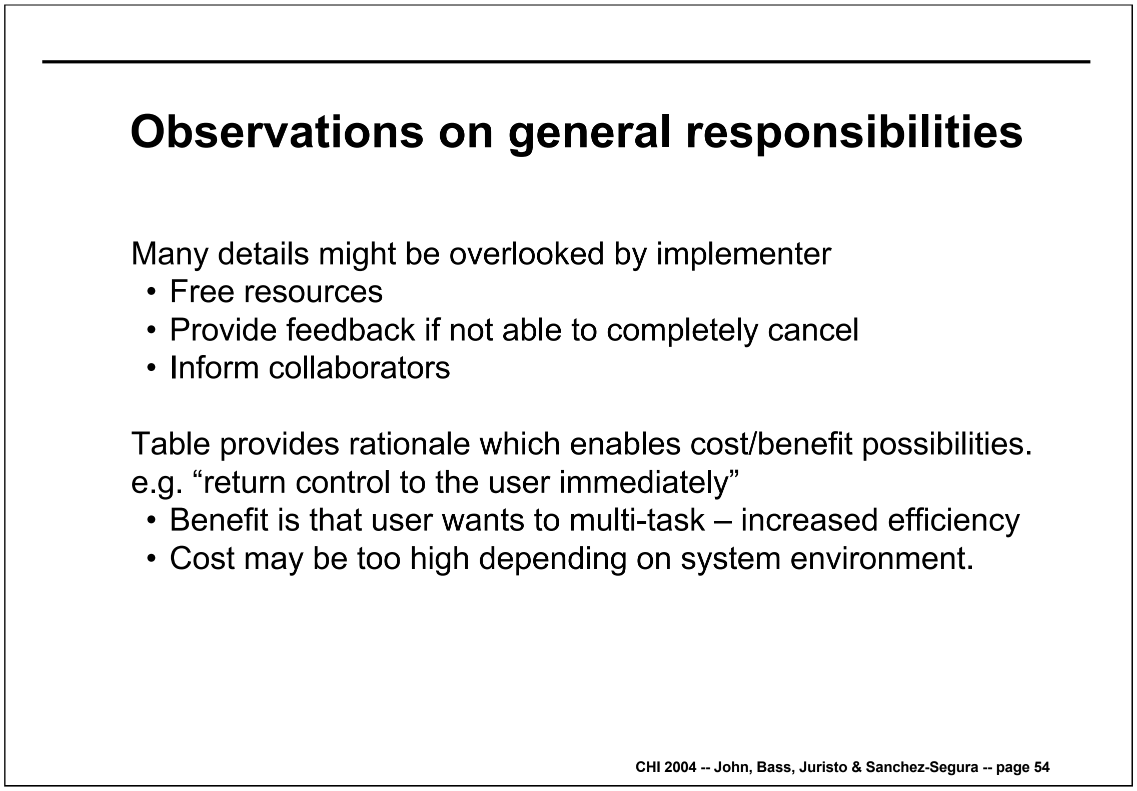
R18. If control cannot be returned to the user, inform the user of this fact (and ideally, why that is the case)

R19. Estimate the time it will take to cancel within 20%

R20. Inform the user of this estimate.

> If the estimate is between 1 and 10 seconds, changing the cursor shape is sufficient.
>
> If the estimate is more than 10 seconds, and time estimate is with 20%, then a progress indicator is better.
>
> If estimate is more than 10 seconds but cannot be estimated accurately, consider other alternatives (see TN, footnote 8)

R21. Once the cancellation has finished the system must provide feedback to the user that cancellation is finished, e.g., if cursor was changed to busy indicator, change it back to normal; if progress bar was displayed, remove it; if dialog box was provided, close it.



Observations on general responsibilities

Many details might be overlooked by implementer

- Free resources
- Provide feedback if not able to completely cancel
- Inform collaborators

Table provides rationale which enables cost/benefit possibilities.
e.g. “return control to the user immediately”

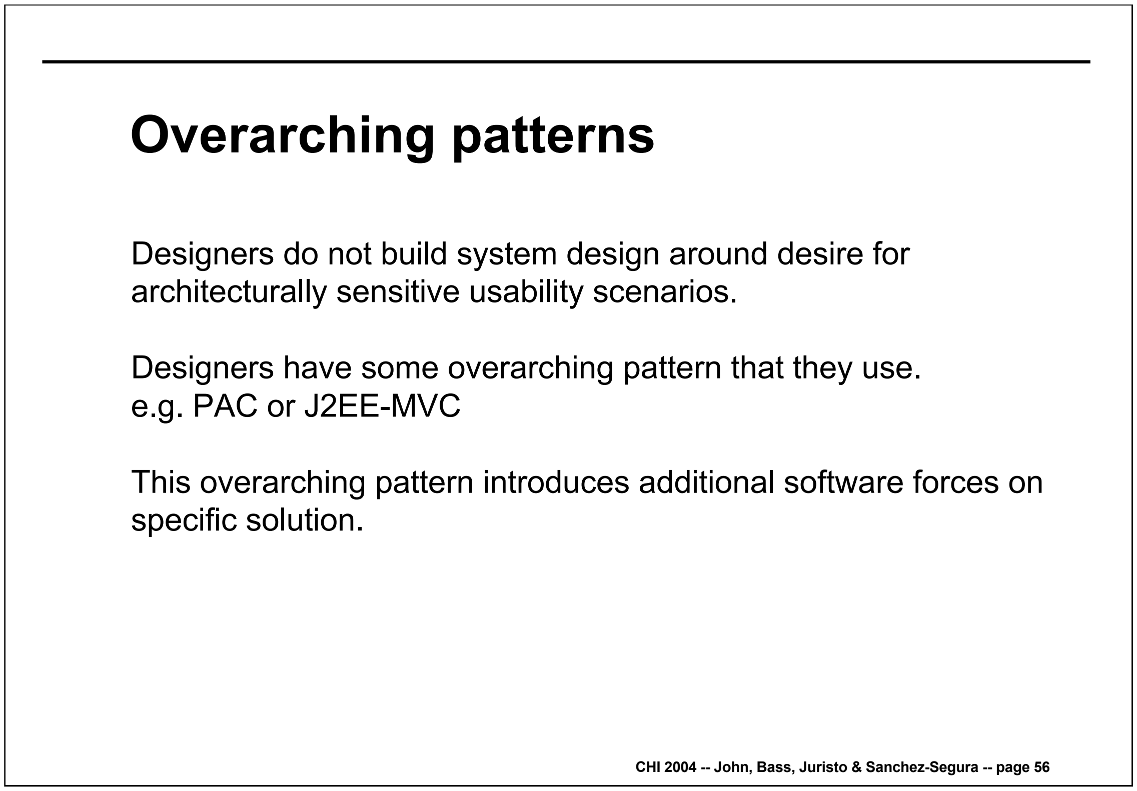
- Benefit is that user wants to multi-task – increased efficiency
- Cost may be too high depending on system environment.

CHI 2004 -- John, Bass, Juristo & Sanchez-Segura -- page 54

Small Group Exercise:

Apply the problem statement and forces to a real-world problem brought by the members of the breakout group. (one group per instructor)

Report-out to the entire group

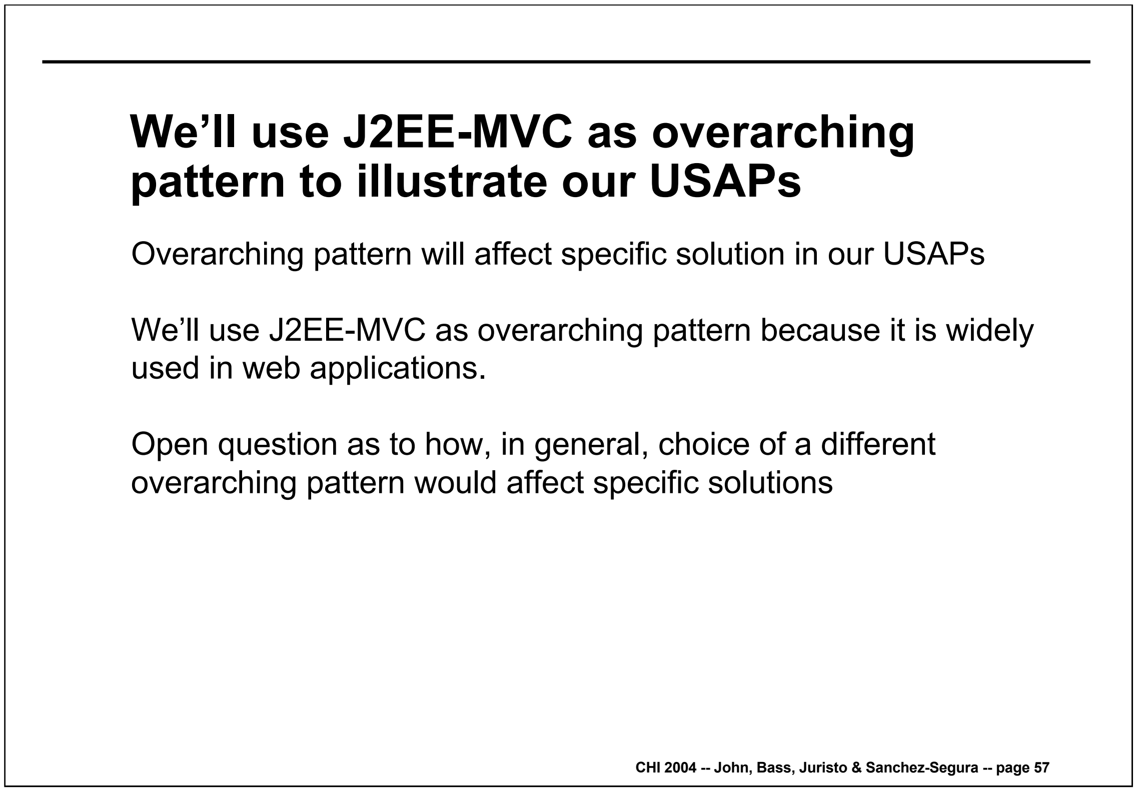


Overarching patterns

Designers do not build system design around desire for architecturally sensitive usability scenarios.

Designers have some overarching pattern that they use.
e.g. PAC or J2EE-MVC

This overarching pattern introduces additional software forces on specific solution.



We’ll use J2EE-MVC as overarching pattern to illustrate our USAPs

Overarching pattern will affect specific solution in our USAPs

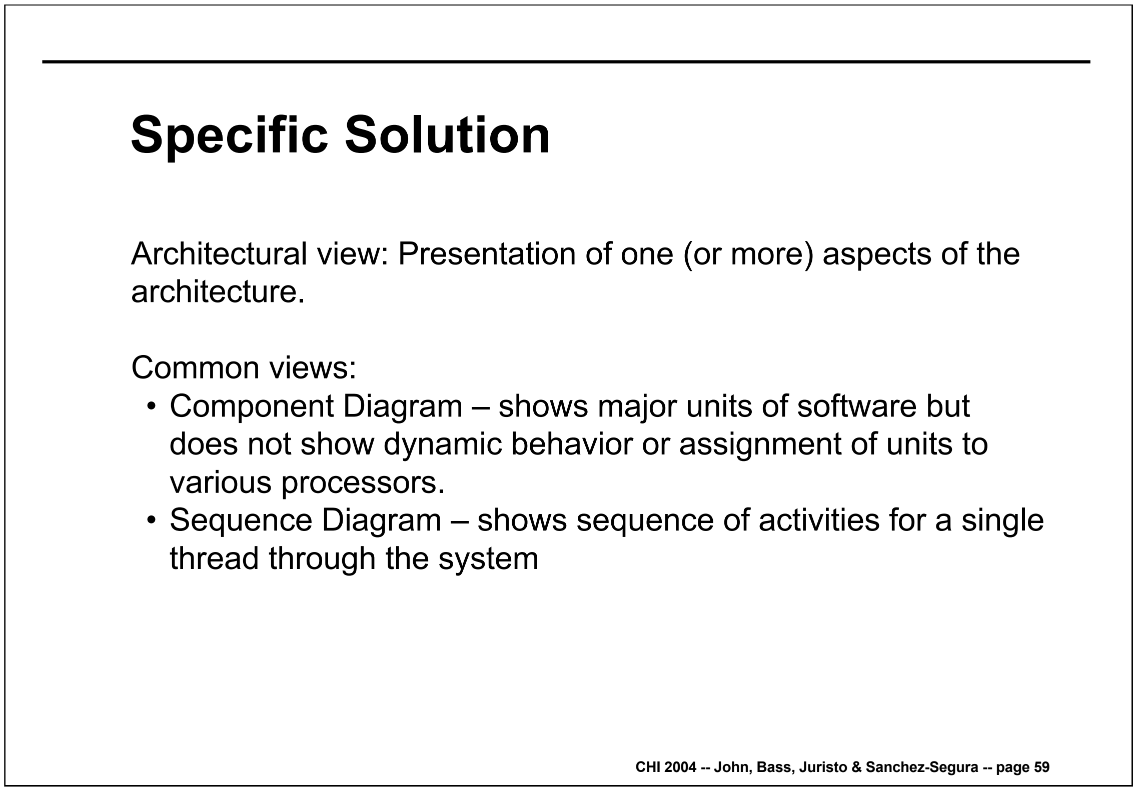
We’ll use J2EE-MVC as overarching pattern because it is widely used in web applications.

Open question as to how, in general, choice of a different overarching pattern would affect specific solutions

We'll use a non-critical task for the example

This implies that

- The user can have control while the cancellation is happening
- The user need not acknowledge the results of the cancellation



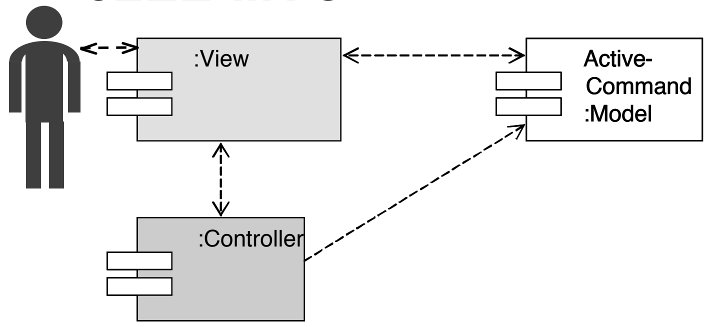
Specific Solution

Architectural view: Presentation of one (or more) aspects of the architecture.

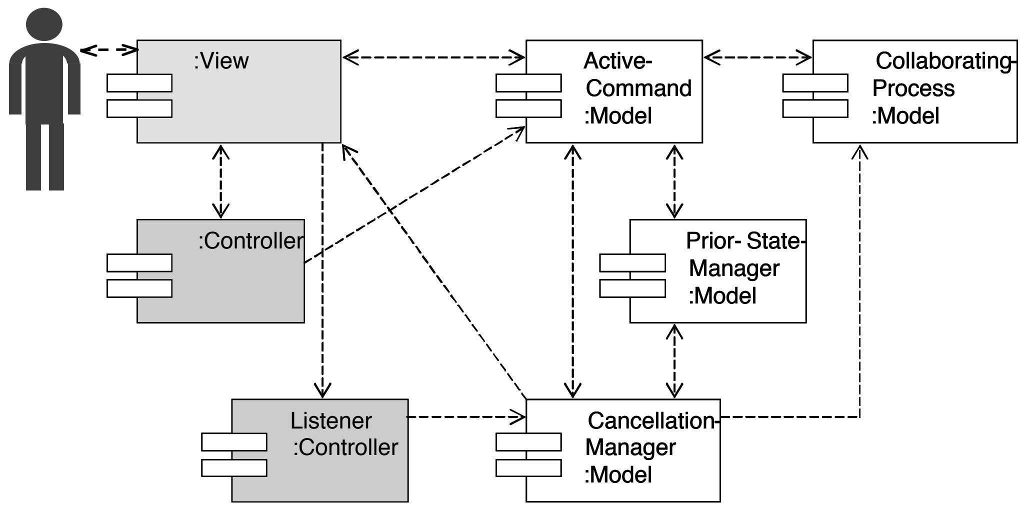
Common views:

- Component Diagram – shows major units of software but does not show dynamic behavior or assignment of units to various processors.
- Sequence Diagram – shows sequence of activities for a single thread through the system

Context of the specific solution: J2EE-MVC



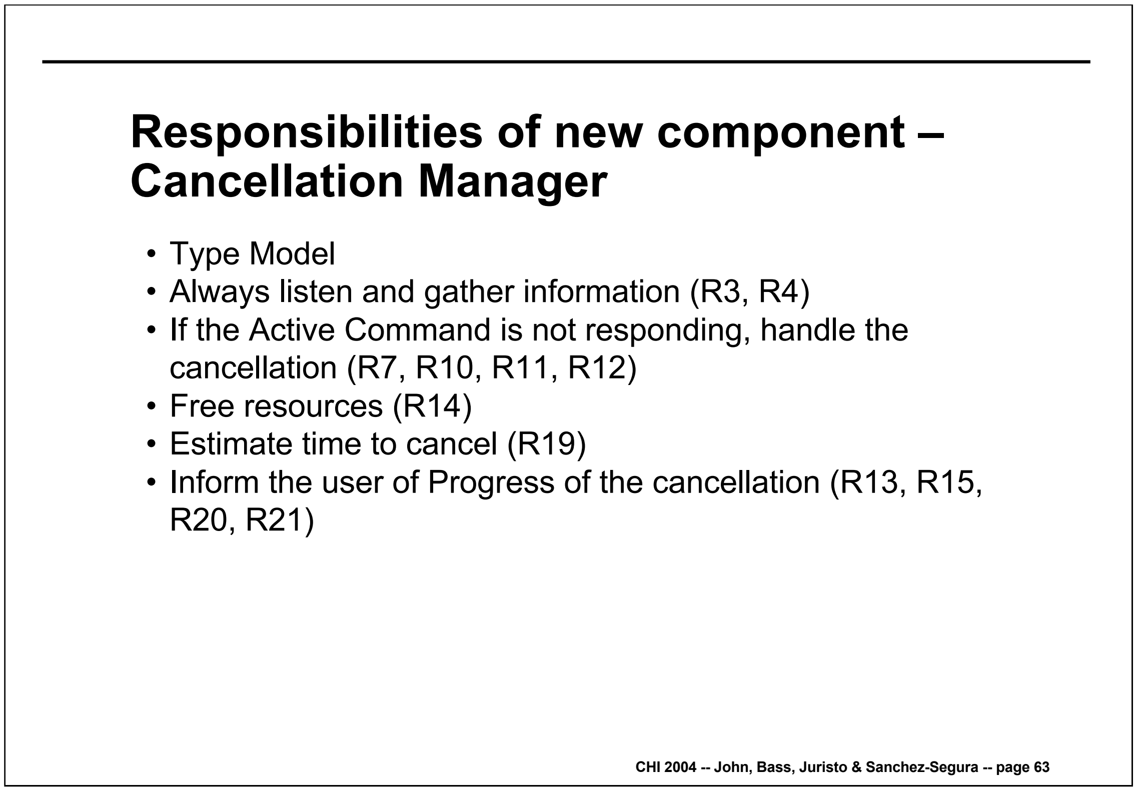
Component diagram for a specific solution to Cancel


:View
Active-Command :Model
Collaborating-Process :Model
:Controller
Prior- State-Manager :Model
Listener :Controller
Cancellation-Manager :Model

CHI 2004 -- John, Bass, Juristo & Sanchez-Segura -- page 61

Responsibilities of new component – Listener

- Type Controller
- Must always listen for the cancel command or environmental changes (R3)



Responsibilities of new component – Cancellation Manager

- Type Model
- Always listen and gather information (R3, R4)
- If the Active Command is not responding, handle the cancellation (R7, R10, R11, R12)
- Free resources (R14)
- Estimate time to cancel (R19)
- Inform the user of Progress of the cancellation (R13, R15, R20, R21)

CHI 2004 -- John, Bass, Juristo & Sanchez-Segura -- page 63

Full text of responsibilities assigned to the Cancellation Manager in this example solution

R3. Must always listen for the cancel command or environmental changes

R4. Must always gather information (state, resource usage, actions, etc.) that allow for recovery of the state of the system prior to the execution of the current command

R7. An active portion of the application must ask the infrastructure to cancel the command,

If R7, then R10. The collaborating processes have to be informed of the cancellation of the invoking command (these processes have their own responsibilities that they must perform in response to this information, possibly treat it as a cancellation.). The information given to collaborating processes may include the request for cancellation, the progress of cancellation, and/or the completion of cancellation.

If R7, then R11. Restore the system back to its state prior to execution of the command. OR R12. Restore the system back to as close to the state prior to execution of the command as possible

If R12, then R13. Inform the user of the difference between the prior state and the restored state.

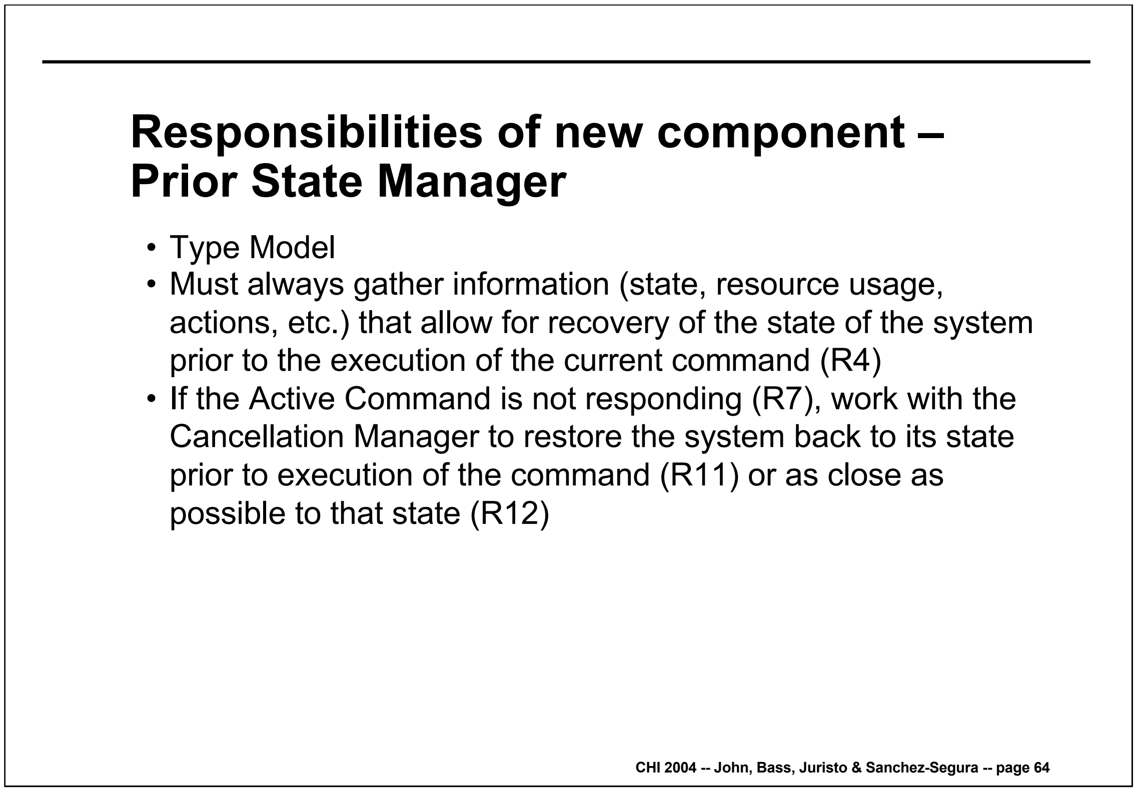
R14. All resources that can be freed must be freed.

If any resources are not capable of being freed, then R15. Inform the user of the partially-restored resources in a manner that they will see it.

R19. Estimate the time it will take to cancel within 20%

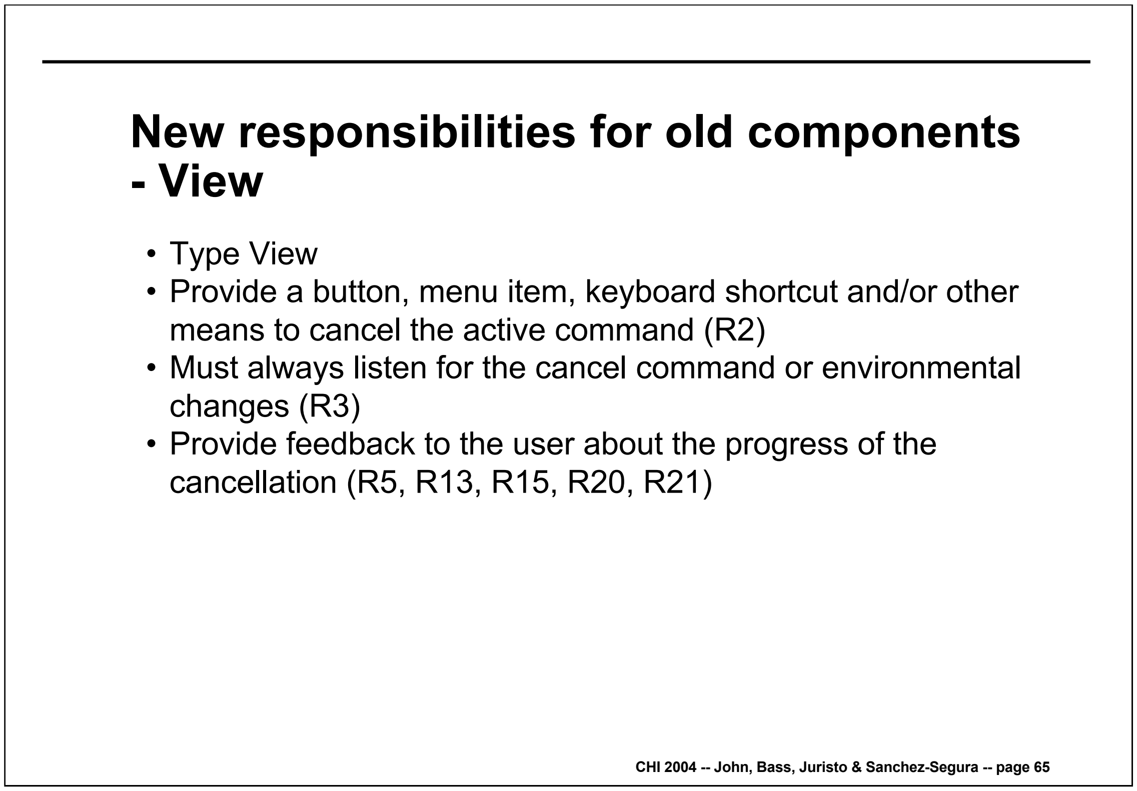
R20. Inform the user of this estimate.

R21. Once the cancellation has finished the system must provide feedback to the user that cancellation is finished, e.g., if cursor was changed to busy indicator, change it back to normal; if progress bar was displayed was displayed, remove it; if dialog box was provided, close it.



Responsibilities of new component – Prior State Manager

- Type Model
- Must always gather information (state, resource usage, actions, etc.) that allow for recovery of the state of the system prior to the execution of the current command (R4)
- If the Active Command is not responding (R7), work with the Cancellation Manager to restore the system back to its state prior to execution of the command (R11) or as close as possible to that state (R12)



New responsibilities for old components - View

- Type View
- Provide a button, menu item, keyboard shortcut and/or other means to cancel the active command (R2)
- Must always listen for the cancel command or environmental changes (R3)
- Provide feedback to the user about the progress of the cancellation (R5, R13, R15, R20, R21)

CHI 2004 -- John, Bass, Juristo & Sanchez-Segura -- page 65

Full text of responsibilities assigned to the View in this example solution

R2. Provide a button, menu item, keyboard shortcut and/or other means to cancel the active command

R3. Must always listen for the cancel command or environmental changes

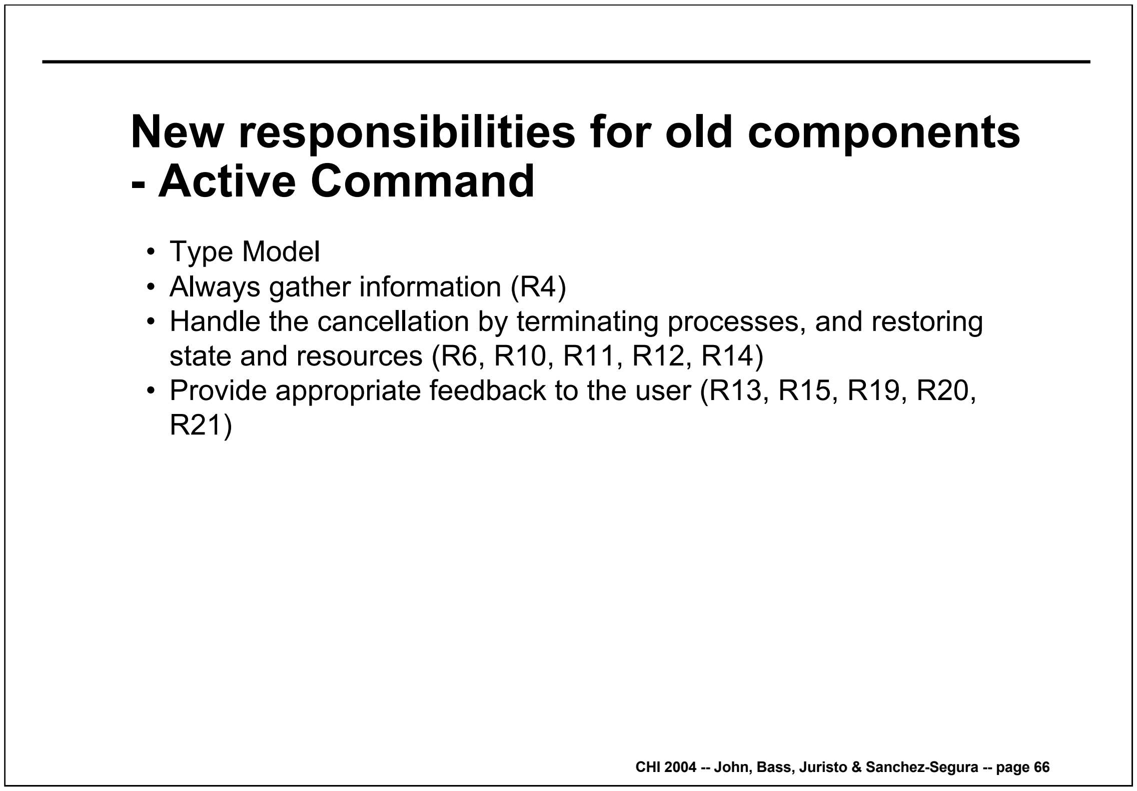
R5. Must acknowledge receipt of the cancellation command appropriately within 150 msec.

If any module did R12, then R13. Inform the user of the difference between the prior state and the restored state.

If any module did R14, then R15. Inform the user of the partially-restored resources in a manner that they will see it.

R20. Inform the user of the time estimate.

R21. Once the cancellation has finished the system must provide feedback to the user that cancellation is finished, e.g., if cursor was changed to busy indicator, change it back to normal; if progress bar was displayed was displayed, remove it; if dialog box was provided, close it.



New responsibilities for old components - Active Command

- Type Model
- Always gather information (R4)
- Handle the cancellation by terminating processes, and restoring state and resources (R6, R10, R11, R12, R14)
- Provide appropriate feedback to the user (R13, R15, R19, R20, R21)

CHI 2004 -- John, Bass, Juristo & Sanchez-Segura -- page 66

Full text of responsibilities assigned to the Active Command in this example solution

R4. Must always gather information (state, resource usage, actions, etc.) that allow for recovery of the state of the system prior to the execution of the current command

R6. The command must respond by canceling itself (I.e., it must fulfill Responsibilities R10 to R21 (e.g., an object-oriented system would have a cancel method in each object)

If R6 then R10. The collaborating processes have to be informed of the cancellation of the invoking command (these processes have their own responsibilities that they must perform in response to this information, possibly treat it as a cancellation.). The information given to collaborating processes may include the request for cancellation, the progress of cancellation, and/or the completion of cancellation.

If R6, then R11. Restore the system back to its state prior to execution of the command. Or R12. Restore the system back to as close to the state prior to execution of the command as possible

If R12, then R13. Inform the user of the difference between the prior state and the restored state.

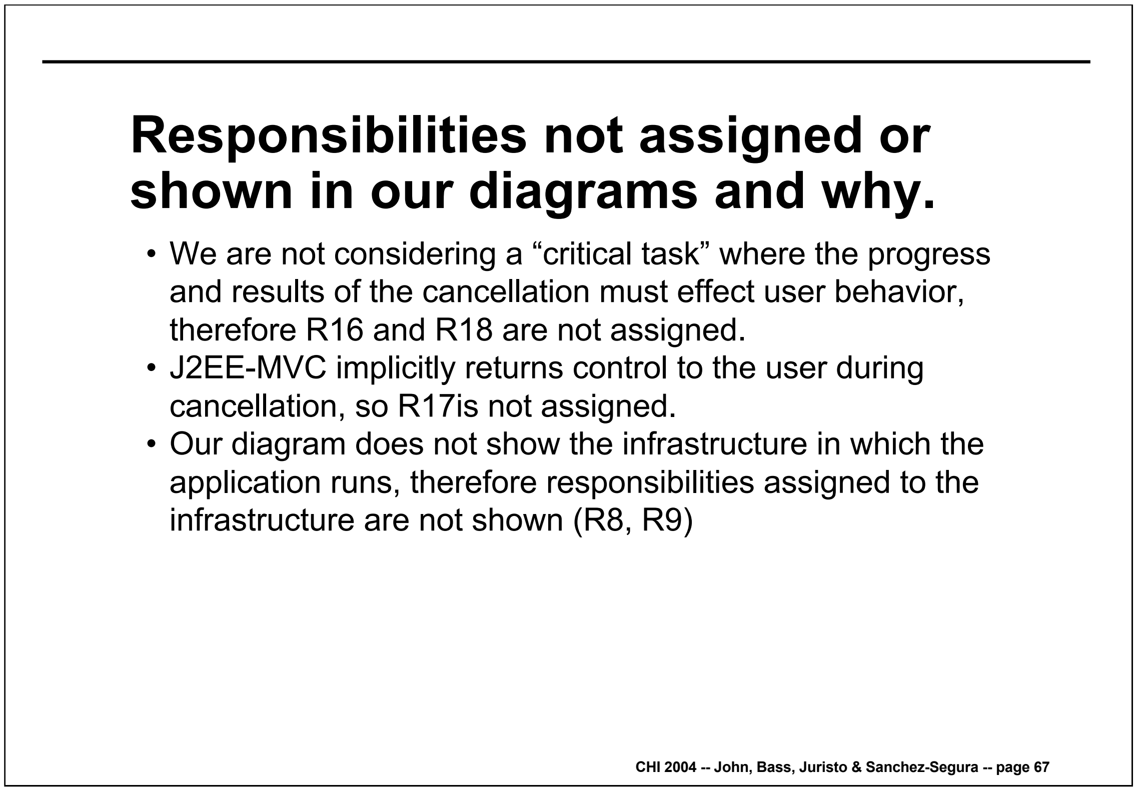
R14. Resources that can be freed must be freed

If any resources are not capable of being freed, then R15. Inform the user of the partially-restored resources in a manner that they will see it.

R19. Estimate the time it will take to cancel within 20%

R20. Inform the user of this estimate.

R21. Once the cancellation has finished the system must provide feedback to the user that cancellation is finished, e.g., if cursor was changed to busy indicator, change it back to normal; if progress bar was displayed was displayed, remove it; if dialog box was provided, close it.



Responsibilities not assigned or shown in our diagrams and why.

- We are not considering a “critical task” where the progress and results of the cancellation must effect user behavior, therefore R16 and R18 are not assigned.
- J2EE-MVC implicitly returns control to the user during cancellation, so R17is not assigned.
- Our diagram does not show the infrastructure in which the application runs, therefore responsibilities assigned to the infrastructure are not shown (R8, R9)

List of responsibilities not assigned to our components or not shown in the diagrams.

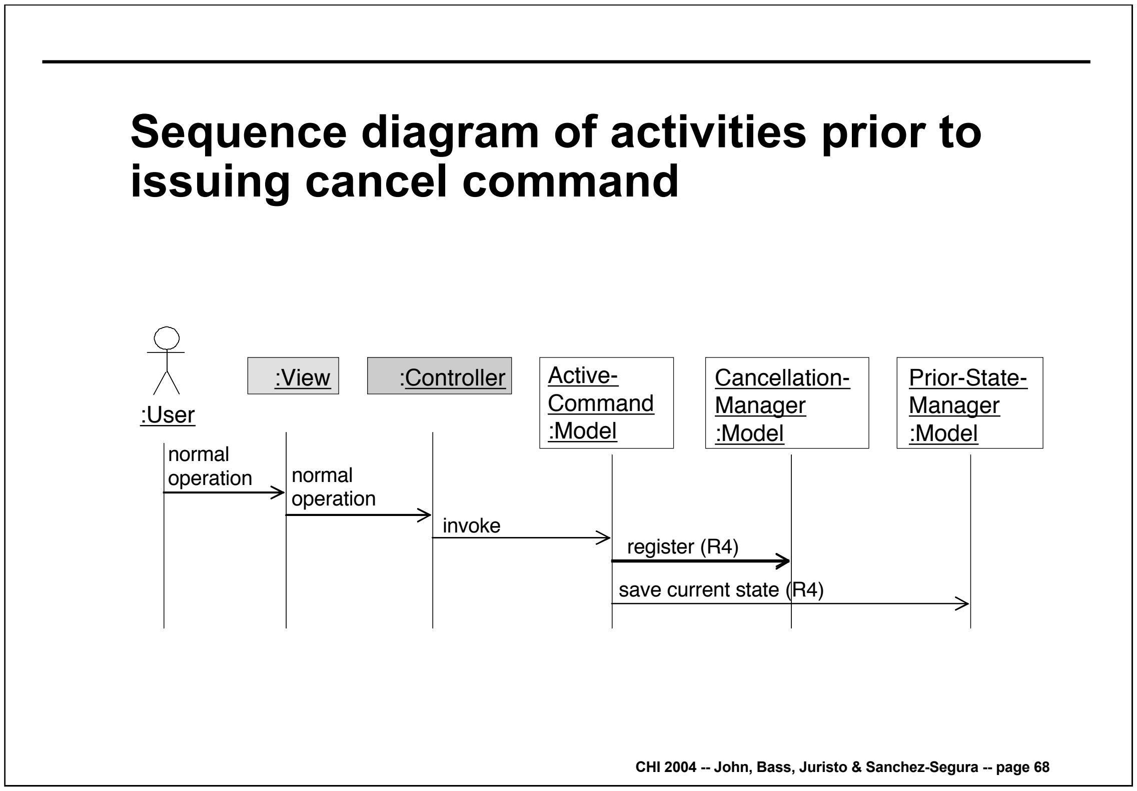
R8. The infrastructure itself must provide a means to request the cancellation of the application (e.g., task manager on Windows, force quit on MacOS)

R9. If either R7 or R8, then the infrastructure must have the ability to cancel the active command (I.e., it must fulfill Responsibilities R10 to R21)

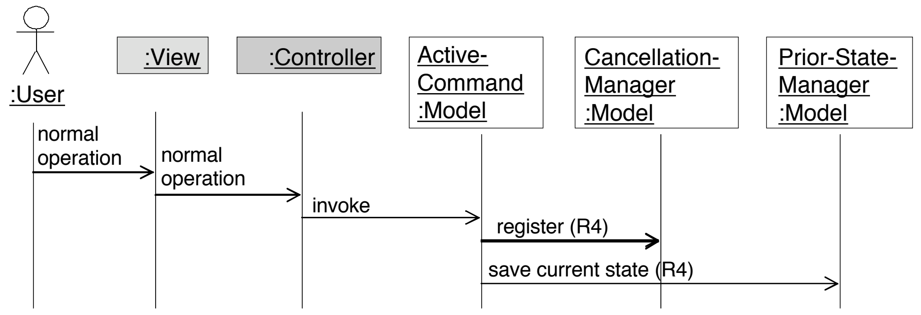
R16. Require acknowledgement from the user that they are aware of the partially-restored nature of the cancellation. (we’re not doing a “critical task” in this example)

R17. Return control to the user, or not, depending on the forces from the task (implicit in J2EE-MVC)

R18. If control cannot be returned to the user, inform the user of this fact (and ideally, why that is the case) (we’re not doing a “critical task” in this example)



Sequence diagram of activities prior to issuing cancel command

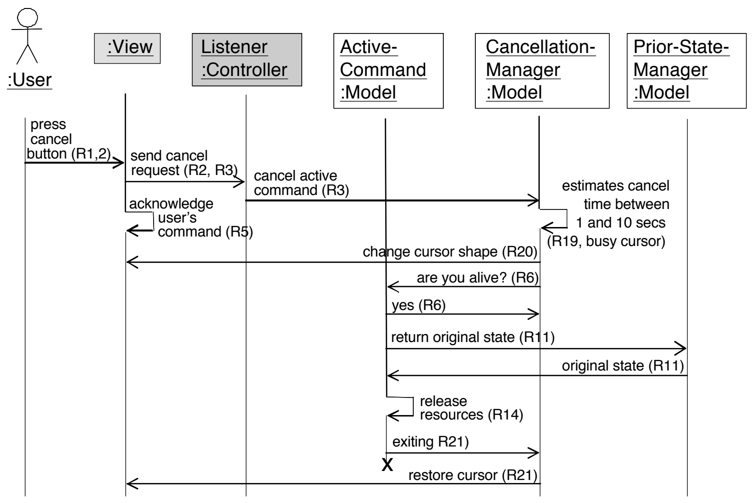


CHI 2004 -- John, Bass, Juristo & Sanchez-Segura -- page 68

Comments about Sequence Diagram

Only components that participate in this sequence are shown.

Sequence diagram of activities after issuing cancel command


:User
:View
Listener :Controller
Active-Command :Model
Cancellation-Manager :Model
Prior-State-Manager :Model
press cancel button (R1,2)
send cancel request (R2, R3)
cancel active command (R3)
acknowledge user's command (R5)
estimates cancel time between 1 and 10 secs (R19, busy cursor)
change cursor shape (R20)
are you alive? (R6)
yes (R6)
return original state (R11)
original state (R11)
release resources (R14)
exiting R21)
X
restore cursor (R21)

CHI 2004 -- John, Bass, Juristo & Sanchez-Segura -- page 69

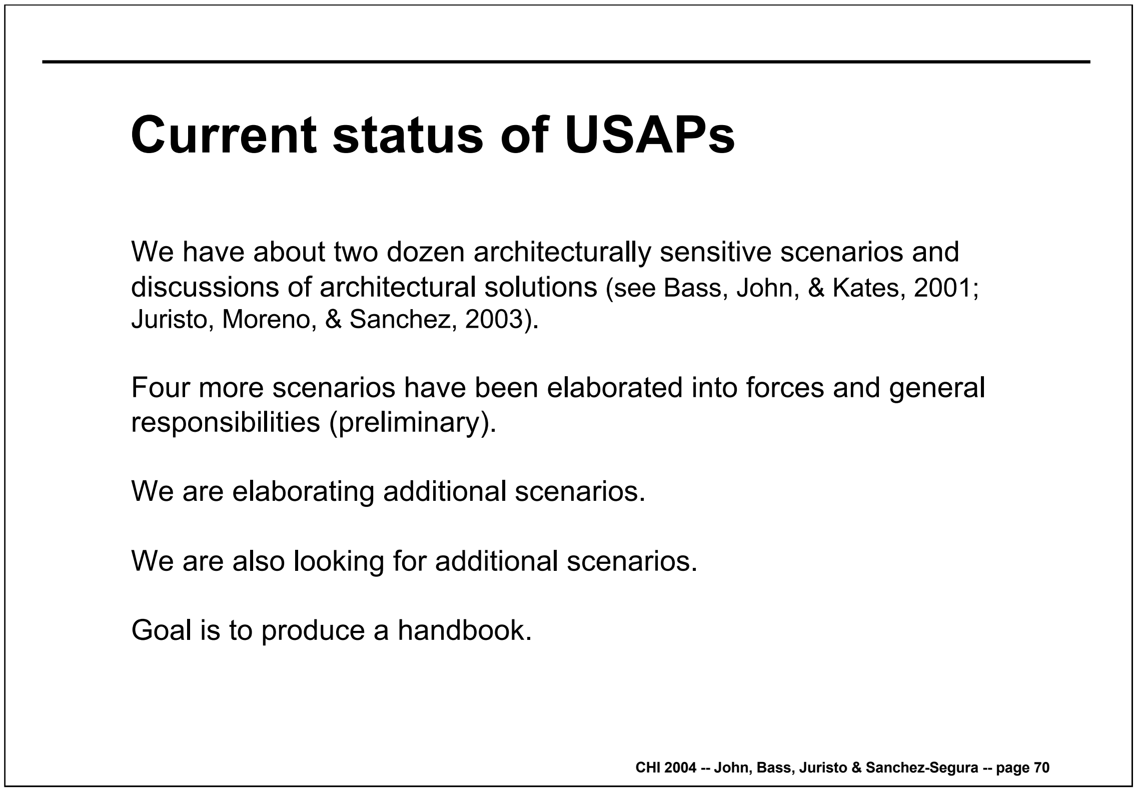
Comments about Sequence Diagram

Only components that participate in this sequence are shown.

An important portion of cancel is that the Listener is on separate thread of control (otherwise listener may be blocked because command is not responding and command owns the active thread).

Sequence diagram does not make this explicit. It is implicit in fact that the Listener responds regardless of state of active command.

Sequence diagram is UML (standard). Difficult to show threads in UML.



Current status of USAPs

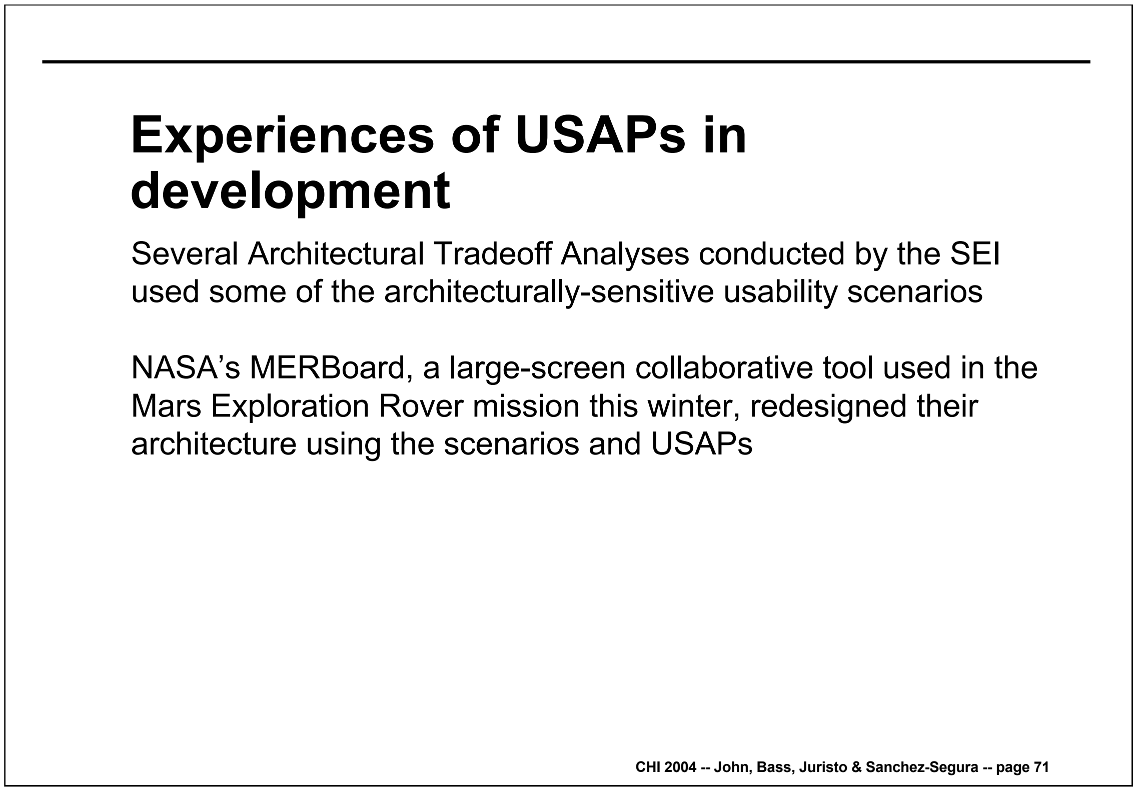
We have about two dozen architecturally sensitive scenarios and discussions of architectural solutions (see Bass, John, & Kates, 2001; Juristo, Moreno, & Sanchez, 2003).

Four more scenarios have been elaborated into forces and general responsibilities (preliminary).

We are elaborating additional scenarios.

We are also looking for additional scenarios.

Goal is to produce a handbook.



Experiences of USAPs in development

Several Architectural Tradeoff Analyses conducted by the SEI used some of the architecturally-sensitive usability scenarios

NASA's MERBoard, a large-screen collaborative tool used in the Mars Exploration Rover mission this winter, redesigned their architecture using the scenarios and USAPs

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.

Tutorial Summary

Software architectural design can support iterative design through separation based patterns, but some usability issues are difficult to resolve through iterative design.

Architecturally sensitive scenarios are examples of problems that are difficult to implement once architecture is designed.

USAPs are an attempt to capture some of these problems, provide rationale to support cost/benefit analysis, provide general set of responsibilities for any solution, and provide sample specific solution to further guide software designer.

Currently have about two dozen architecturally sensitive scenarios and are in process of turning these into USAPs.

Questions?

To keep up with recent developments concerning usability and software architecture (U&SA), check our website periodically: www.UandSA.org

Appendix I
General Usability Scenarios

(excerpt of Bass, L., John, B. E., & Kates, J. (2001). Achieving usability through software architecture (CMU/SEI-2001-TR-005). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University)

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.

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. 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.

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.

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)

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. 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.)

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.

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.

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.

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.

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.

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.

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.

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.

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.

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 on-line 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)

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.

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.

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.

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.

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).

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.

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.

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)

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)

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.

27. Supporting Personalization
(not in CMU/SEI-2001-TR-005)

A user wants to work in a particular configuration of features that the system provides. The user may want this configuration to persist over multiple uses of the system (as opposed to having to set it up each time). Systems should enable a user to specify their preferences for features and provide the possibility for these preferences to endure. For example, customizing Netscape’s toolbar or saving a hierarchical structure of bookmarks.

Appendix II
Details of the Usability Benefit Hierarchy

(excerpt from Bass, L., John, B. E., & Kates, J. (2001). Achieving usability through software architecture (CMU/SEI-2001-TR-005). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University)

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

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.

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.

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.

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.

[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.

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.

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

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

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.

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 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).

Appendix III: Usability Supporting Architectural Pattern Template

Bonnie E. John, Len Bass, Maribel Sanchez-Segura, and Rob J. Adams, Sept. 2004

Name: Usability Supporting Architectural Patterns must have suggestive names, which give an idea of the problem addressed and the solution in a word or two.

Usability Context
The context is divided into three parts: the situation (or general usability scenario), the conditions under which this situation is relevant, and the potential benefits to the users if the problem arising from this situation is solved.

Situation: A brief description of the situation that makes this pattern useful from the user's viewpoint.

Conditions: Any conditions concerning the situation constraining when the pattern is useful

Potential Usability Benefits: A brief description of the benefits to the user if the solution is implemented. We use the usability benefit hierarchy taken from Bass & John to express these benefits.

Problem
The problem is expressed as the requirements arising from human desires and capabilities and the task versus the constraints deriving from the state of the software or the environment. These forces result in responsibilities that the software must fulfil to solve the problem. The responsibilities are part of the solution, but it is valuable to see how they arise. They are, therefore, presented here.

Forces exerted by the environment and task	**Forces exerted by human desires and capabilities**	**Forces exerted by the state of the software system**	**Responsibilities of the general solution**
This field describes the state of the environment and the task, which may include issues of task criticality or control, among others.	This field describes the human desires and needs, which may include issues of salience or control, among others.	This field describes the state of the software, which may include issues of resources and control, among others.	This field contains the responsibility of the software and an argument about why this responsibility is necessary given the user/task needs and/or the system/environment constraints.

Specific Solution
The specific solution is situated in a partial design that reflects requirements that the designer believes are more important than cancel. The designer will make particular choices of overall architecture, MVC is one example, we need to present our table as an example of how the general solution is customised for the overall architecture.

General responsibilities of the software	**Forces exerted by previous design decisions**	**Allocation of responsibilities to specific components**	**Rationale**
This field describes the general responsibilities of the software identified in the problem description.	This field describes the forces that come from design decisions taken independently of usability aspects but that influence the way in which these aspects must be designed.	This field describes the responsibilities of the software components involved in the design of the usability aspects.	Justification of how this allocation of responsibilities to specific modules satisfies the problem.

Components diagram of specific solution

Sequence diagram of specific solution

Deployment diagram of specific solution (if necessary)

Appendix IV
Relationship of Forces to General Responsibilities for Cancelling Commands

Bonnie E. John, Len Bass, Maribel Sanchez-Segura, Rob Adams, Elsa Golden

19-Feb-2004

Forces exerted by the environment and task	Forces exerted by human desires and capabilities	Forces exerted by the state of the software	Responsibilities that must be satisfied by any software design solution:
Networks are sometimes unresponsive.	Users slip or make mistakes, or explore commands and then change their minds, but do not want to wait for the command to complete.	Software is sometimes unresponsive	R1. Must provide a means to cancel a command
Sometimes changes in the environment require the system to terminate			
	Users have to communicate their intentions to the software through overt acts (e.g., finger movements)	Software has to receive an action from the user to do something	R2. Provide a button, menu item, keyboard shortcut and/or other means to cancel the active command.
No one can predict when the environment will change	No one can predict when the users will want to cancel commands		R3. Must always listen for the cancel command or
			R4. Must always gather information (state, resource usage, actions, etc.) that allow for recovery of the state of the system prior to the execution of the current command

	User needs to know that the command was received within 150 msec, or they will try again. It can be assumed that a user is looking at a button as they click it. People can see changes in color in their fovea. People can see changes in intensity in their peripheral vision.			R5. Must acknowledge receipt of the cancellation command appropriately within 150 msec. The acknowledgement must be appropriate to the manner in which the command was issued. For example, if the user pressed a cancel button, changing the color of the button will be seen. If the user used a keyboard shortcut, flashing the menu that contains that command might be appropriate.
The task or environment has indicated that the command should stop (e.g., the OS has determined that there is not enough memory to continue)	User has communicated a desire for the command to stop	EITHER	The command itself is responsive at the time of cancellation	R6. The command must respond by cancelling itself (I.e., it must fulfill Responsibilities R10 to R21 (e.g., an object-oriented system would have a cancel method in each object)
		OR	The command itself is not responsive at the time of cancellation	R7. An active portion of the application must ask the infrastructure to cancel the command, or
				R8. The infrastructure itself must provide a means to request the cancellation of the application (e.g., task manager on Windows, force quit on MacOS)

				R9. If either R7 or R8, then the infrastructure must have the ability to cancel the active command with whatever help is available from the active portion of the application (I.e., it must fulfill Responsibilities R10 to R21)
		The command has invoked collaborating processes		R10. The collaborating processes have to be informed of the cancellation of the invoking command (these processes have their own responsibilities that they must perform in response to this information, possibly treat it as a cancellation.). The information given to collaborating processes may include the request for cancellation, the progress of cancellation, and/or the completion of cancellation.
	User wishes to operate the system as if their command had not been issued.	EITHER	the system is capable of rolling back all changes to the state prior to execution of the command.	R11. Restore the system back to its state prior to execution of the command.
		OR	the system is not capable of rolling back some of the changes made during the operation of the command prior to cancelation	R12. Restore the system back to as close to the state prior to execution of the command as possible
				R13. Inform the user of the difference between the prior state and the restored state.

The system should remain stable over time (if there are resources not returned, it may lead to subsequent system crash, e.g., memory leak)	User wants the software to return to the pre-command state.		R14. All resources that can be freed must be freed
	Users need to know to what degree this desire was achieved (because it may effect what they do in their current or future tasks)	IF some resources has been irrevocably consumed and cannot be restored	R15. Inform the user of the partially-restored resources in a manner that they will see it.
For critical tasks, the inability to restore state or resources may require external actions to acknowledge the partial restore	Users may not always be paying attention, or may forget to take action		R16. Require acknowledgement from the user that they are aware of the partially-restored nature of the cancellation.
EITHER it is not critically important to the task that the cancellation be complete before another action can be taken	User wants to multi-task depending on the time it will take to cancel the command, typically more than 1 second.	The command takes more than 1 second to cancel	R17. Return control to the user, or not, depending on the forces from the task
	Users expect accurate feedback (within 20%, see TN) so they can plan their multitasking.		R18. If control cannot be returned to the user, inform the user of this fact (and ideally, why that is the case)
			R19. Estimate the time it will take to cancel within 20%
OR it is critically important to the task that the cancellation be complete before another action can be taken			R20. Inform the user of this estimate. If the estimate is between 1 and 10 seconds, changing the cursor shape is sufficient. If the estimate is more than 10 seconds, and time estimate is with 20%, then a progress indicator is better.

			If estimate is more than 10 seconds but cannot be estimated accurately, consider other alternatives (see TN, footnote 8)
	User wants to be notified when the cancellation has finished		R21. Once the cancellation has finished the system must provide feedback to the user that cancellation is finished, e.g., if cursor was changed to busy indicator, change it back to normal; if progress bar was displayed was displayed, remove it; if dialog box was provided, close it.

References

References in Usability and Software Architecture

Bass, L. J. & John, B. E. (2000) Achieving Usability Through Software Architectural Styles. *Extended Abstracts of CHI, 2000* (The Hague, The Netherlands, April 1-6, 2000) ACM, New York. pp. 502-509.

Bass, L., John, B. E. (2002) Supporting the CANCEL command through software architecture, CMU/SEI-2002-TN-021. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.
http://www.sei.cmu.edu/publications/documents/02.reports/02tn021.html

Bass, L. J. & John, B. E. (2003) Linking usability to software architecture patterns through general scenarios*. Journal of Systems and Software, 66*(3), 187-197.

John, B. E. & Bass, L. J. (2001) Usability and software architecture*. Behaviour and Information Technology, 20*(5), 329-338.

Bass, L., John, B. E. & Kates, J. (2000) *Achieving usability through software architecture (CMU/SEI-2001-TR-005)*. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.
http://www.sei.cmu.edu/publications/documents/01.reports/01tr005.html

Juristo , N., Moreno, A. M., & Sanchez, M. (2003) *Deliverable D.3.4. Techniques, patterns and styles for architecture-level usability improvement*. - ESPRIT project (IST-2001-32298)
http://www.ls.fi.upm.es/status/results/deliverables.html

References in software engineering and software architecture

Bachmann, F., Bass, L., Chastek, G., Donohoe, P., & Peruzzi, F. (2000) *The architecture based design method (CMU/SEI-2000-TR-001)*. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.
http://www.sei.cmu.edu/publications/documents/00.reports/00tr001.html

Bass, L.; Clements, P. & Kazman, R. (2003). *Software Architecture in Practice*. 2nd edition. Reading, MA: Addison Wesley Longman.

Buschmann, F., Meuneir, R, Rohnert, H., Sommerlad, P. and Stal, M., (1996) *Pattern-Oriented Software Architecture, A System of Patterns*, Chichester, Eng: John Wiley and Sons.
Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., & Stafford J., (2003) *Documenting Software Architectures: Views and Beyond*, Addison Wesley.

Clements, P., Kazman, R, & Klein, M. (2001). *Evaluating software architectures: Methods and case studies.* Boston: Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., (1995) *Design Patterns, Elements of Reusable Object-Oriented Software*, Reading, Ma: Addison Wesley Longman.

Hillside Group: Information about patterns and pattern languages can be found at `http://www.hillside.net/` `(19 February 2004).`

J2EE-MVC is documented at
`http://java.sun.com/blueprints/patterns/MVC-detailed.html`
`(19 February 2004).`

Klein, M. & Bachmann, F. (2000). *Quality Attribute Design Primitives* (CMU/SEI-2000-TN-2000-017). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.
`www.sei.cmu.edu/publications/documents/00.reports/00tr017.html`

Laprie, J.-C. (1992) *Dependability: Basic Concepts and Terminology*. Springer-Verlag: Vienna.
McCall*,* J. (2001) Quality Factors. In *Encyclopedia of Software Engineering* (2nd edition) John Marciniak, ed., John Wiley, New York, pp 1083-1093
Smith, C. & Williams, L., (2001) *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Reading, Ma.:Addison Wesley Longman.

References in human performance usability

Beyer, H. & Holtzblatt, K. (1998) *Contextual Design*. San Francisco, CA: Morgan Kaufmann Publishers, Inc.

Card, S. K., Moran, T. P. & Newell, A. (1983) *The Psychology of Human-Computer Interaction*. Hillsdale, NJ: Erlbaum.

Gram, C. & Cockton, G. (1996) *Design Principles for Interactive Systems*. London, England: Chapman and Hall.

Newell, A. & Simon, H. A. (1972) *Human Problem Solving*. Englewood Cliffs, NJ: Prentice-Hall.

Newman, W. & Lamming, M. (1985) *Interactive System Design*. Wokingham, England: Addison-Wesley Publishing.

Nielsen, J. (1993) *Usability Engineering*. Boston, MA: Academic Press Inc.

Shneiderman, B. (1998) *Designing the User Interface*, 3rd ed. Reading, MA: Addison-Wesley.